Player/Stage
Player driver implementation for ERSP

Project at DIKU's robotlab. Winter 2006

Bue Petersen <buep@diku.dk>
Jonas Fonseca <fonseca@diku.dk>

Department of Computer Science
University of Copenhagen
Winter 2006

Contents

List of Figures

1 Introduction

In this project we enable DIKU's[1] *Scorpion robots* from *Evolutions Robotics*[1] to be programmed through *Player*, an open framework for programming robots and sensor application. The robots are normally programmed using the *Evolution Robotics Software Platform* (*ERSP*), which have several limitations, technical as well as practical. *Player*, developed by *The Player Project*[5], has been proposed by students, scientific staff, and other users at DIKU as a more suitable platform for DIKU. One of the benefits of using Player is *Stage*, a 2D simulator that allows robot control programs to be run on models of real-world robots in a simulated world.

1.1 An Overview of Player/Stage

Player is a *network server* for robot and sensor control and runs on the robot/sensor hardware or on the computer connected to the hardware. In our case the Scorpion robots are connected through USB cable to a laptop running the server. The *Player server* provides access over the IP network to the robot sensors and actuators through the *interfaces* the *Player driver* implements for that specific piece of hardware (eg. a Scorpion robot).

The *robot control program* (interchangeably also called *user program*, control program or just program) acts as a *client* and talks to the Player server over a TCP socket. The client *subscribe interfaces*, optionally configure the hardware, and then starts reading data from sensors and writing command to actuators.

The distributed nature of Player allows that user programs can be written in any language that supports TCP sockets. There already exist client-side utilities and libraries in common languages like C, C++, Java, and Python.

Two Player drivers for different robots that implements the same interface will allow a user program to control both kind of robots. E.g. if the Pioneer2 and Scorpion robot both supports the interface for driving a robot in 2D the same program could control both robots.

The Player server allows multiple devices to present the same interface as in the example and it supports that multiple user programs (clients) subscribe the same interfaces. Clients can even connect to several Player servers. This gives possibilities like distributed sensing and control or controlling a robot from one program and logging the robots sensor data from another.

Player together with the 2D simulator Stage, referred to as *Player/Stage*, could simulate a population of robots moving and sensing in a 2D world. In the context of this project it is interesting that a user program could drive a simulated robot as well as a real robot with no or only minor changes.

Stage present various *sensor models* and odometry that is used the same way as on the real robots. Stage presents Player interfaces for the *Stage devices*, so the sensor models or odometry could be used as if it was a real robot.

1.2 Project Goals

Our primary purpose is to be able to run a simple robot control program on the Scorpion robots using, as a minimum, odometry, i.e. driving, turning, setting speed, and reading

[1] Department of Computer Science University of Copenhagen, `http://www.diku.dk`

velocity and acceleration. This should be done through the Player framework instead of the ERSP framework provided by Evolution Robotics.

The overall secondary purpose is to enable other students at DIKU to use Player/Stage in the programming of the robots. Our work should be available to the students attending courses using the robot laboratory at DIKU. "Available" here means that we should provide drivers and documentation for using Player/Stage instead of ERSP. This means we have to:

- Make Player drivers for as many of the robots sensors and interfaces as possible. This will enable more advanced programming of the robots.
- Document how to make new drivers for the robots. This may include implementing new sensors and describing interfaces for Player. (**Writing Player/Stage Drivers**)
- Make a **User Manual** to other students describing how to use our implementation. This description should be a supplement to the official Player documentation.
- Evaluate if it is possible to use Stage to prototype a simple control program for the robot. This includes comparing results from running the prototype in Stage and on the physical robots.

Our test and experiment with Player/Stage will be done with a simple robot control program. It will be necessary to write a robot description for the simulator to have the Stage simulator mimic the Scorpion robots as precise as possible. The result of this task/process will effect the outcome of the comparison between using Stage and the physical robot.

Besides these result, we will try to make a short study of how our work could be used and benefit the future work with DIKU's robots. We will analyse what new possibilities the Player framework will provide and how students at DIKU can benefit from using the Player framework to share code with other students and researchers around the world. We will also do experiments and describe problems and limitations related to using the Player framework at DIKU's robot lab.

1.3 Motivation

Since the robot lab at DIKU acquired new robots in the autumn of 2005, students have experimented with the ERSP software bundled together with the robots. This software generally provides a high level of abstraction, which makes it very easy to develop software for the robots. The abstraction level, however, has also proven to be a weakness since it puts certain limitations on the possibility to experiment and explore robot systems that require capabilities that are not provided by ERSP. Also, the ERSP software is closed source, which restricts the ability to study and modify essential parts of the platform, or integrate custom pieces of software together with the ERSP platform.

The Player framework has been proposed as a replacement for the current platform. It has several qualities that address the technical and practical limitations related to the existing platform. We will give a short comparison below that highlights some of these qualities we find that Player has compared to ERSP.

Player is is an open platform and available under an Open Source license. This is of great significance as students will be able to develop and share their project with other people outside DIKU as they will be developing on a common platform that is used by universities

around the world. Additionally, with the support for simulations using Stage or Gazebo it will be possible for students to develop and test their system without being requiring access to the physical robots and using one of the limited number of ERSP licenses.

We would like to investigate further whether moving to Player framework on the Scorpion robots is feasible.

1.4 Related Work

There already exists a Player driver for the ER1 and ERSDK robot platforms offered by the same company that makes the Scorpion robots. It takes the low-level approach and operates the robot by communicating with it directly using a serial communication over the USB link. The driver is far from complete as it only supports the `position2d` interface, it has, however, been integrated into the Player CVS repository. It is developed by David Feil-Seifer, who claims that the driver will also work for the Scorpion robots. No one at DIKU has been able to support this claim.

1.5 Audience and Reading

We assume the reader of this document is familiar with the basic concepts of robotics and concurrency, as well as fundamental ideas in distributed systems. Experience with robotics, such as programming of a robot and implementing algorithms for robot control etc., will be an advantage. No knowledge of Player/Stage or ERSP and the Scorpion robots are required in advance. Readers not familiar with either may want to refer to Section 2 and 3 respectively before reading the rest of the document.

The user manual in Section 8 and the tutorial for writing Player drivers provided in Appendix A are primarily written for students and users of the robot lab at DIKU. People outside of DIKU may however also find them useful.

The material covered in Section 4 and 5 requires some knowledge about how drivers in Player work. The reader may want to refer to the Player driver tutorial in Appendix A to get a quick introduction.

1.6 Resources

As part of this project, we have developed a source package containing among other things a Player driver. A wiki page has been used for keeping track of the project in its early state. Finally, video has been recorded of experiments with the robots. More information on these resources are given below.

If there is a CD-rom included with this document it will contain this document, the source package and the video archive respectively in folder named:
`erspplayerdriver` and `erspplayerdriver-video`.

ERSP Player Driver Source Package

The software developed as part of this project is contained in the *ERSP Player Driver Source Package* in the following also referred to as the *source package*. The source package contains README files that documents the various parts of the source package and should help to get you started. An overview of the source package is given in Section 5.10.

You can obtain the source package from DIKU ImageLab's SVN repository. The repository can be browsed from `http://image.diku.dk/svn/robot/`. **The direct link for the source package is:** `http://image.diku.dk/svn/robot/trunk/erspplayerdriver/`.

To use the software package and the implemented drivers requires that Player version 2.0.3 (or greater) and Stage version 2.0.1 (or greater) have already been installed on the local system and that the source package has been checked out on the local system. Additionally, the ERSP 3.1 software (Evolution Robotics Software Platform) must be installed to use the Scorpion robots.

Project Wiki Page

We have created a **wiki page** for this project at DIKU ImageLab's wiki: `http://image.diku.dk/mediawiki/index.php/Robotprojekter_2006:_Player_Stage`

On the wiki page, you will find links for downloading this report, our tutorials, and documents that are part of our project goals. An archived version of the source package (as of February 28th) will also be available. Finally, you will be able to find links for interesting articles and other information.

Experiment Video Archive

We have made a little **video archive** available. It contains small movie-clips documenting some experiments with the robots. The videos in the archive are all DivX 5.2.1 encoded avi-files.

The video archive is available from the DIKU ImageLab's SVN repository under the name `erspplayerdriver-video`. The repository can be browsed from `http://image.diku.dk/svn/robot/`. **The direct link for the source package is:** `http://image.diku.dk/svn/robot/trunk/erspplayerdriver-video/`.

2 Player/Stage Details

Below we describe some specific details about the Player/Stage framework that will motivate our design decisions for the driver implementation. Although, we start with some general details, only a limited part of the Player framework is documented here based on what we find relevant for discussions in later sections. Refer to the official Player website[5] for further details and to find out how to program a robot using Player/Stage.

2.1 The Player/Stage Architecture

There are 3 key concepts in Player[3]:

Interface: A specification of how to interact with a certain class of hardware like one type of a robots sensors or actuators. The interfaces define the syntax of how to issue commands to actuators and how to read inputs from sensors through messages[2]. Typically an interface handles only one type of sensors or actuators. For example, a `position2d` interface exists that handles odometry sensors and motor commands for the robot - together considered the hardware for the robots position i 2D.

Driver: The software that talks with the actual hardware and translates its input and output to conform to one or more relevant interfaces. The driver hides hardware specific details and makes that piece of hardware appear to be the same as any other hardware of the same type by providing the interface in question. For example, every infrared range sensor will be used the same way, that is through the `ir` interface, in a user program.

Device: A device is the topmost abstraction in Player for the hardware and used through a fully-qualified *device address*. All messaging in Player occurs among such devices and through the interfaces.

As an example of how the concepts relate, the main topic of this project is to develop a driver that provides a number of interfaces so, once the driver is instantiated, devices are available for robot control programs. A visualization is given in Figure 1.

Drivers, devices, and interfaces are also important when configuring Player as we will see in Section 2.6 below.

2.2 Multiple Servers and User Programs

Player uses a client-server architecture that is fully distributed. Among other things, it means that user programs can connect to one or more Player servers from multiple locations. This multiplicity of connectivity and topology poses several challenges when initializing hardware and controlling subscriptions in a Player driver. We will return to this topic later and only now state some common situations that can arise.

First, consider the cases with a single user program and Player server. They can both be run on the same host. This is normally the case for simulation, but is also possible for the setup at DIKU, where the laptops are powerful enough to run both the server and user program. However, some situations may require that they run on separate hosts. Many

[2] See Section 2.5

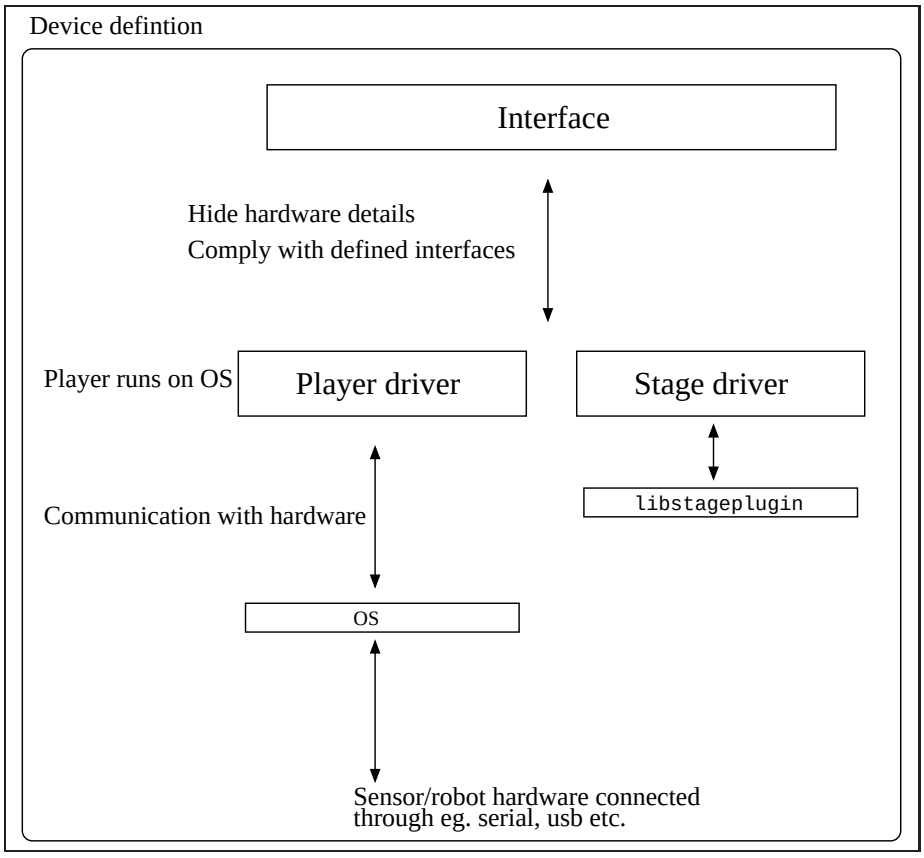

Figure 1: **Visualization of the 3 key concepts of Player.** When writing a user program the device definition is used as a fully-qualified address to access sensors and actuators that belong to that class. The device is the abstraction for hardware that conforms to an interface and controls the hardware through the driver that does the most of the work. The Stage simulator is also shown and is accessed through a plug-in and gives access to some of the same interfaces that real hardware through their driver may conform to. Device examples are `position2d:0`, `laser:0` or `odometry:::position2d:1`, and a driver could be the `ersp`, `p2os` or `khepera`. The interface that is part of the device definition `position2d:0` would then be the `position2d` interface.

robots run an embedded operation system with limited resources. In this case, the robot might only be able to run the Player server, whilst the user program is developed and finally run on a separate host and simply connects to the server over the network.

The above case of a single user program and server can be expanded to include multiple Player servers. Consider an example, where a user program will control more than one robot in a collaborating robots experiment. This would be easy with Player as user programs can connect to several Player servers. In other words, the single user program can create multiple clients–one to control each robot. Since it is the only user program, it has exclusive access to every robot, neither the driver nor the user program will have to consider any use-cases not already covered in the case with a single server and user program.

Finally, there is the case of multiple user programs and one or more Player server. As before, this setup allows collaboration to take place via the framework provided by Player. Contrary to the previous case, user programs are required to synchronize access to hardware resources among themselves. One way to accomplish this is to only allow one user program

to commands the actuators of a robot at a time, while the rest of the user programs may subscribe to sensor data message. In other words, each user program can only control devices on its own robot but is allowed to snoop on its neighbors' devices.

2.3 Drivers

Player is distributed with a rich set of drivers for commonly used robots and robotics related hardware. However, not all robots and hardware are supported "out of the box". In this case, Player has a rich framework for writing drivers for new hardware. Two methods exists for extending Player with new drivers. A driver may either be built-in at compile time or plugged-in at runtime. The choice of whether to implement a built-in or plugin driver mainly depends on whether the driver is going to be submitted for inclusion in the coming version of Player. Drivers that are going to be maintained out-of-tree as a separate package, must be plugins. The ability to extend Player by dynamically load drivers into a running Player server offers a great level of flexibility when developing and using drivers.

2.3.1 Virtuel Drivers

In the above, we describe drivers as a method for providing access to sensors or actuators, but in Player it is also possible to expose algorithms through *virtual drivers*. This driver type offer a powerful way to extend Player with advanced and reusable features.

A virtual driver implements special purpose functionality in Player through one or more interfaces. Instead of deriving input from real hardware, they usually build upon the interfaces provided by one or more other drivers. A virtual driver can get data from other drivers, process and use advanced algorithms on the data, and finally output the result to the interfaces it provides. Besides the interfaces, usually targeted for hardware drivers, a set of interfaces exist purely for these virtual drivers. There are no restrictions regarding which interfaces they can provide, i.e. a virtual driver can also provide interfaces normally associated with hardware, such as an odometry interface. Using this approach, virtual drivers can be used to apply error correction to hardware derived data considered prone to errors.

A good example of a virtual driver is the `acml`-driver based on the Adaptive Monte-Carlo Localization algorithm. This virtual driver takes as input odometry and laser data, and provides two new interfaces that estimates the robot's position based on the algorithm's computation on the input data.

2.4 Devices and Device Addressing

The Player server makes it possible for drivers to provide multiple devices all using the same interface. For example, a driver can provide two `position2d` devices, each with different origin of data, or two separate drivers can each provide a `position2d` that originates from two different robots. To allow drivers and clients to easily distinguish between devices, each device that is provided by a Player server must be given a unique device address. This is usually done in a configuration file passed to the Player server when starting up.

The fully-qualified device address consists of 5 parts: *key*`:`*host*`:`*robot*`:`*interface*`:`*index*. Broken down, the *key* is a unique name, the *host* and *robot* fields are usually the host name and port number on which the Player server listens but may vary on the context[4], the *interface* is the interface name of the device, and the *index* is a interface specific sequence number. In the

most common usage only the interface and index needs to be specified, e.g. `position2d:0` and `ir:1`.

However, when multiple devices with the same interface type are provided by a driver, a *key* value must also be given. The reason is that the *index* part is determined dynamically by the driver when it registers its devices at the server. It, therefore, needs a hard-coded and unique way to distinguish devices of the same interface type. Since the *host* and *robot* parts of the device addresses are usually assumed to be the address and port number of the Player server in which the driver is loaded, they can be left as the empty string. For example, a driver providing front and back IR sensors as two different `ir` devices can use: `front:::ir:0` and `back:::ir:1`.

With regards to drivers, all of the above is only of concern during initialization, since internally the device address will simply be a tuple consisting of the interface ID and the device index. Contrary for user programs, mainly the device index is of concern, since the interface type will be given explicitly in the datastructures and objects used by the program.

2.5 The Message Passing System

As mentioned above, all commanding of actuators and reading of sensor data in Player is done by messaging (message passing). In other words, the core system of Player is a queue based message passing system. In the basic message flow, clients (user programs) are mainly consumers that subscribe to messages from a set of device addresses and drivers are producers that push, primarily, sensor data to the clients.

The clients and drivers communicate using a request-reply oriented protocol. Each driver has a single incoming message queue and can publish messages to specific clients in response to requests. E.g. a message is sent to the driver to ask for sensor data or command an actuator. A reply will then return the sensor reading to the client.

A driver can also publish messages to the incoming queue of other drivers or it can broadcast to all subscribed clients. The interface specifications defines the semantics for a set of messages that belongs to that interface. The Player core library is responsible for passing the messages.

2.5.1 Message Types

The message passing system has 3 basic types of messages.

Data messages are mainly used by drivers to publish sensor readings and other changes in device state, such as motor stalls.

Command messages are sent by clients to order a driver to change the state of a specific device it controls. A command may set a new velocity for a motor or specify that a device should turn off its power.

Configuration messages provide a method for clients to configure device properties as well as query the static geometry information, such as the poses of individual sensors or the dimensions of a bumper. The former gives control over various device hardware settings, while the latter makes it possible for clients, such as visualizers, to adapt to the geometry of a specific device.

Each of the above message types are used for giving messages a generic label. They are each extended with multiple subtypes. This allows drivers to filter first on the message type and later on the interface specific subtypes.

2.5.2 Message Queues and Data Modes

As already mentioned, each client and driver has a single incoming message queue. The queue has a limited size and may therefore over time run full if messages are not periodically processed. Usually, drivers will process its message queue as the last part of the main driver loop if the queue is not empty. The processing usually involves manually filtering and matching which messages the driver supports.

For user programs, message processing is hidden away by the API and done implicit by the client object when requested by the program. The user program will normally request this in the start of its read-think-act loop to perform the "read" part. A user programs may choose to deviate slightly from this convention and use sleep to control the duration of commands issued to devices. Such programs can easily cause the above problem of queue overflow to occur.

To accommodate this, Player defines two data modes: *pull* and *push*, as well as the concept of replace rules[2]. The idea is to enable clients to notify the server whether they want the server to send all messages as soon as possible (pull) or whether they want to only have it sent messages when requested (pull). This allows the clients to control the flow of messages to best suit their needs. On top of this replace rules can be configured so that newer messages with a specific type or origin will always replace already queued messages of the same type. Combined, a client can ensure that its queue will never overflow and that it will always receive the newest data messages by setting the data mode to pull and configure a replace rule for all data messages.

2.6 Player and Stage usage

When using Player/Stage, three programming steps are required to make a robot control program. First, a Player client must be created. During creation, the client will establish a connection to a Player server with a given host name and port number. Second, the client must subscribe to the devices it will be using. For each device, this is done by creating a proxy object that represent the device. Third, a read-think-act loop must be implemented. It will be the actual control program for the robot. In the loop, the client should periodically process incoming messages to update the proxy objects with information from messages containing sensor data.

Before writing a robot control program, the configuration file for the Player server should be written. The configuration file tells the server, which drivers to initialize along with definitions and settings for each driver. Driver settings are specific to each driver and must contain the name of the driver and a list of the devices the driver provides, and for virtual drivers, which devices they require at startup. Additionally, driver settings can be used to configure the behavior of the driver, such as how the driver should connect to the hardware, i.e. what USB or serial port to use.

Mostly, a user program will be bound explicitly to certain device types. This affects portability of user programs between different hardware platforms and the simulator, since the poses of sensors might not be the same. Some of the issues can be addressed by using

device configuration requests to adapt to different hardware. This can, however, be non-trivial.

2.7 The Stage Simulator

Stage works with Player as a driver, where the driver provides a simulator back-end. From the programmers point of view Player/Stage is used the exact same way as the combination of Player and a real robot. The difference is the configuration file for Player, which specifies that Stage is used by the use of the driver *stage* and the plugin *libstageplugin*. We call it at *Stage Configuration File*, contrary to a Player configuration file, if it configures Stage. A Stage configuration files also describe the models and the environment that Stage will simulate. It is possible in the configuration to specify a model for how one or more real robot should be modelled in Stage and simulated. Especially one must specify which interfaces the robot uses and this would normally be the same that the driver for the real robot implements. An model (like a robot) is modelled from other models in Stage and so is environments like a map.

When modelling a robot (or other object) in Stage the interfaces and predefined models in Stage is used. One is therefore constrained by which interfaces and models Stage support when trying to create a realistic model of a real robot. As an example it will not be possible to describe a robot that uses a bump sensor if the bumper interface in Stage is not supported. It is possible to visualize the robot (you kind of draw the robot), but not program it using the bumper interface.

In section 6 we will get into how the Scorpion robot is modelled and some general features of the Stage simulator.

3 The Scorpion Robots and ERSP

Figure 2: **The Scorpion Robot.** Front-side view of the robot equipped with a laptop computer (Picture from [19]).

This section will briefly describe the Evolution Robotics Scorpion robots[1], for which we will write the Player driver, and the *Evolution Robotics Software Platform* (*ERSP*), upon which we will base the Player driver. The robot is controlled from a (laptop) computer that has the ERSP software installed and is connected to the robot through USB. The robot is able to carry a laptop computer on its "laptop" cradle as depicted in Figure 2.

3.1 The Hardware

The robot is equipped with IR range sensors, a contact bumper and a camera. It has servo powered differential drive on the 2 front wheels and is equipped with a third support wheel behind. The robot has 2 USB connections: one for the robot itself and one for the camera. The robot will presents itself as an Evolution Robotics USB device on a Plug-and-Play capable operation system like Linux and Windows, but it cannot be controlled or used without the ERSP software.

The robots is 44 cm long, 40 cm wide and reaches 42 cm from the floor because of the camera. Its size can be seen in the bounding box picture in Figure 3.

3.1.1 IR Sensor Details

There are 20 IR range finding sensors, where 4 are used for ledge detection to avoid hazardous falls, 3 up-facing sensor for overhang detection to avoid camera damage, and finally, 13 horizontal facing sensors that can be used for obstacle detection.

The 13 horizontal facing sensors are Sharp GP2D12 medium range analog IR sensors. Their range is 10 cm - 80 cm and they have a narrow beam. Although, it is a common robotic and industrial sensor, we have not been able to find out the spread of the beam.

The 7 other sensor are GP2D15 short range digital IR sensors that trigger on distances from 1 cm - 24 cm. As they are binary, they just sense if obstacles are within range or not.

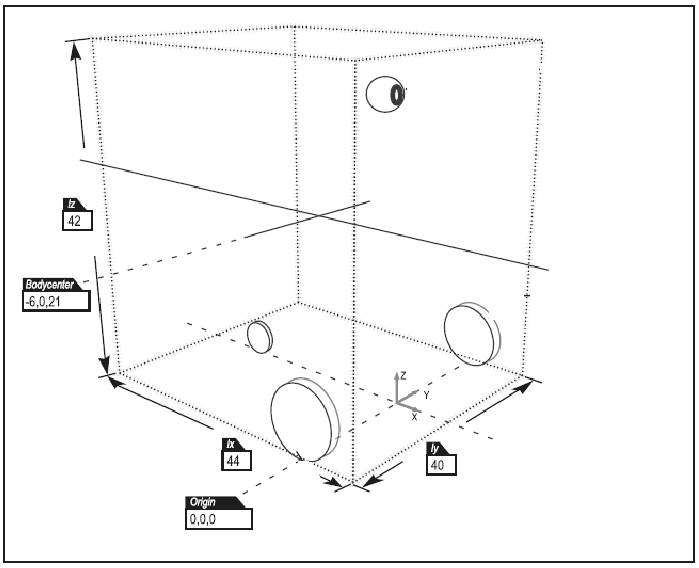

Figure 3: **The bounding box of a Scorpion robot.** The size of the robot is given by the bounding box (Picture from [19]).

Figure 4 shows how the sensors are placed on the robot.

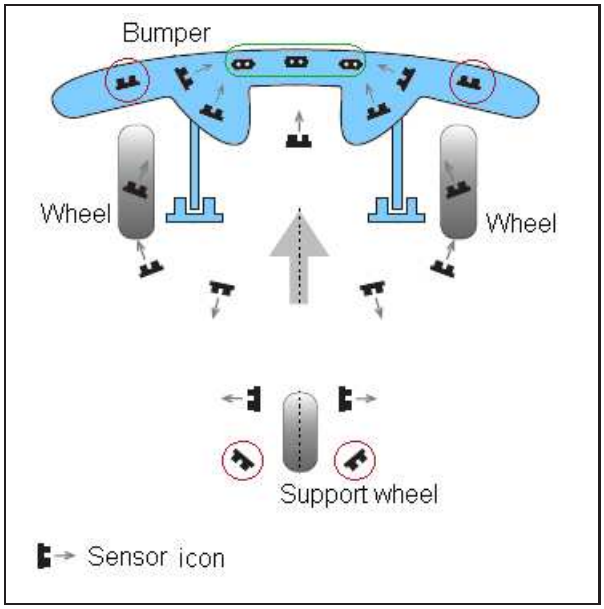

Figure 4: **Sensor on the Scorpion robot.** The figure shows the sensor placement on the robot. The bumper are colored blue. The digital ledge sensors are marked with an red ellipse and the overhang sensors are marked with a rounded rectangle and are placed in the bumper. (Modified picture from [19]).

3.1.2 The Contact Bumper

The robot is equipped with a wide face front bumper to detect collisions. It has low activation force and independent left and right digital contact triggers. The bump sensor is designed to protect the front of the robot and the distance sensor to work undisturbed even when the bumper is activated. Figure 4 shows in blue color the bumper in front of the robot.

3.1.3 The Camera

The camera on the robot is a Logitech Quickcam Pro 3000 USB camera with 640×480 video resolution and 1.3MPixel image resolution. The camera has built-in microphone and support custom lens mount. It is situated in an adjustable camera tilt mount that supports 5 degrees below horizontal to 25 degrees above.

As the camera is standard USB (web) camera it can be accessed directly through the connecting computers OS.

3.1.4 Power Interface, Wheels, and Motors

The robot is controlled by the Evolution Robotics Robot Control Module that handles all motion control, digital and analog sensors, I/O capabilities, sensor I/O, and motor control. The Robot Control Module is accessed through a single USB connection.

The Robot Control Module controls the 12V servo motors with 6.1N-m peak torque and 2000 count encoders for odometry. The robot has 100mm polyurethane wheels which gives a good grip on many surfaces.

There is a 5.2 Ah 12V rechargeable battery that can be monitored through a power interface.

3.2 The ERSP Software

This section will describe features of *ERSP*. We will not go into details with the ERSP software but focus on the relevant parts that we need to know when writing the driver and using the software.

3.2.1 Architecture

ERSP has a layered architecture with mainly 3 layers: *Hardware Abstraction Layer (HAL)*, *Behavior Execution Layer (BEL)*, and *Task Execution Layer (TEL)*. These are supplemented with a set of other *Core Libraries* that make generally useful facilities for robotics applications available.

The Hardware Abstraction Layer is the lowest layer, which is the interface between the robotics software applications and the robotics hardware. In ERSP, the HAL layer is responsible for all the interaction with the robot's sensor and actuators. The Driver Library support the HAL functionality when making programs that uses the robotics hardware at lowest level. Behavior Execution Layer is the middle layer and based upon the HAL layer. Behaviors cover a wide range of functionalities, from reading sensors and driving actuators to higher functionality components such as mathematical operators and algorithms. Eg. Gaussfunctions or Kalmanfilters. Compared to the HAL layer, this layer has a higher abstraction level. Therefore, a programmer using the Behavior Library will be able to program

a robot with behaviors, without having to program the low-level details for controlling the robots, such as the driving system. A behavior could be acting automatic on red and green traffic light. The highest level of abstraction is the Task Execution Layer (TEL) which again is on top of the Behavior Execution Layer. A task express higher-level execution knowledge and coordinate the actions of multiple behaviors. An action that is more appropriate for a task than a behavior is a robot navigating to the kitchen, finding a bottle of beer, and picking it up. Such sequences of task can be programmed through use of the Task Library.[21]

3.2.2 API

Each of the above layers can be accessed by the programmer via libraries: *Driver Library (HAL)*, *Behavior Library (BEL)* and *Task Library (TEL)*. All the functionality that libraries offer is available to the programmer through the ERSP C++ API. We will not go into details with the API that is well documented by Evolution Robotics in the API documentation[18].

There are two obvious approaches to developing robot applications with ERSP. The top-down approach and the bottom-up approach. In a top-down approach as much functionality is used from the top level layer (Task Execution Layer) to reach the programmers goal. When a given functionality in the top level is not implemented the programmer will build it of functionality from the layer just below. In a bottom-up approach the Hardware Abstraction Layer is used to program the robot, but when feasible functionality is used from higher layers. E.g. to spare the programmer for implementing trivial functionality that already exist. One could argue that it would be most effective with the top-down approach, but with higher abstraction levels as in the Task Execution Layer flexibility and generality could be lost.

The minimum functionality needed to make a usable robot control program is access to the robotic hardware. This is done using the Hardware Abstraction Layer to handle and control the hardware.

3.2.3 Accessing the Hardware

All hardware on the Scorpion robot is connected to the Robot Control Module that connects to the computer with ERSP software through a USB connection. All hardware can therefore only be accessed through the ERSP software using the Hardware Abstraction Layer libraries (or higher level functionality). The camera though is stand-alone and could be used without the ERSP software if the operating system supports it.

Initializing and handling the hardware is done with 3 steps:

- First a *Resource Manager* object must be created.
  The Resource Manager determines what resources are available in the current environment, find the appropriate binary drivers and creates C++ object instances of those drivers. Then each bus and its devices are activated and available for use.
- From the Resource Manager a *Resource Container* object can be created.
- To work with the robotics hardware, an *interface* object for the specific piece of hardware must be obtained from the Resource Container object. Through that interface object (now representing the actual hardware) data can be queried or commands sent to it.

The part of the initialization related to obtaining an interface requires explicit naming of the sensor, actuator or other hardware. E.g. it must be done for each sensor used or every driving system. The unique names of the hardware is described in the next section.

The sensors, actuators and other robotic hardware is now accessible in the main part of the program.

Before termination of the program the initialized interfaces should be shut down. This is done by using a release function on the Resource Containers and finally deallocate the Resource Manager itself. [20] [21].

The unique names of sensors available on the Scorpion robot in ERSP is described in Figure 5. These names are used when issuing a request on the Resource Container for an interface.

In ERSP the hardware details is also described in configuration files. *The resource configuration file* (`resource-config.xml`) specifies physical aspect of every piece of hardware on the robot. The file is used often for reference as well as the figure shown. In the file all devices are mentioned and we find it relevant to mention:

Motor devices The two motors on the robot has device ID's `Drive_left` and `Drive_right`.

(Hardware) Avoidance device Device ID is `avoidance` and this device implements, possible in hardware, an default enabled avoidance system using bumper and IR sensors.

Joystick A joystick for Linux is available as device ID `joystick`.

odometry There is a odometry device that report wheel odometry for the differential drive system. Device group ID is `odometry` with members `Drive_right` and `Drive_left`.

3.2.4 Coordinate Systems

ERSP uses either the *robot coordinate system* or the *global coordinate system* to describe the robot's incremental motion and sensor and actuator positions. The robots sensors and actuators position and orientation are described in the documentation with respect to the robot coordinate system. [19]

The robot coordinate system is 3D and has the origo of the system at the floor level, directly below the center of the drive wheels' wheelbase. The positive X-axis of the robot extends forward in the direction the robot is facing and this is also defined as North. When the robot is on a flat surface the positive Y-axis extends West and the positive Z-axis is straight up. The robot's coordinate system is depicted in figure 6.

The world coordinate system is centered where the robot is turned and coincident with the robot's coordinate system at that time. The world coordinate system is fixed to the robot's starting position on the floor, while the robot's coordinate system moves with the robot.

Angles in both coordinate systems is measured from the X-axis and are positive counted counter-clockwise.

The robot's *orientation* is defined from a single angle with respect to the world coordinate system. Beside this it takes three angles to fully specify an object's orientation in 3D:

- **Roll:** The rotation angle around the X-axis
- **Pitch:** The rotation angle around the Y-axis

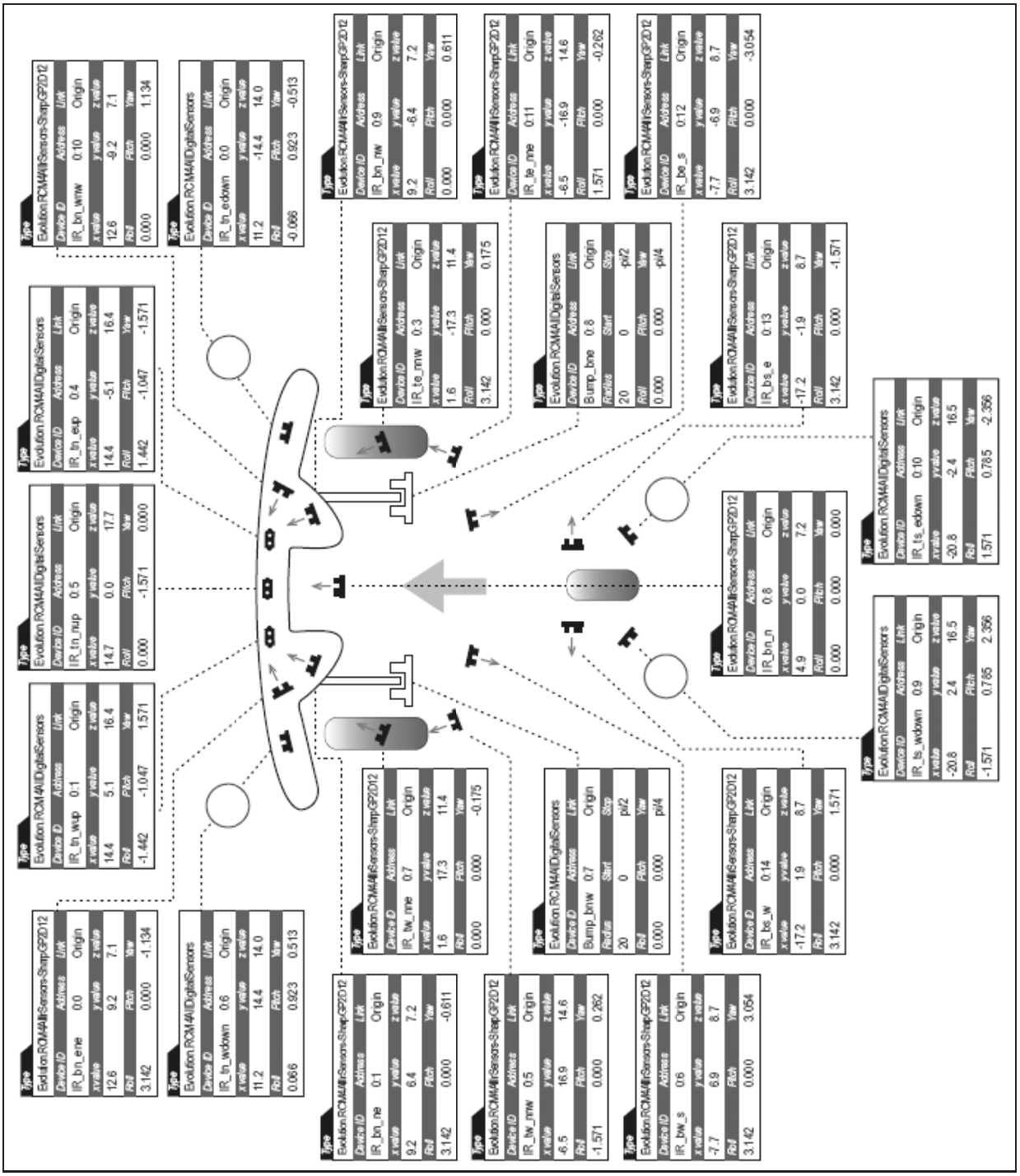

Figure 5: **ERSP sensors on the Scorpion robot.** These unique names are used when requesting access to hardware in ERSP. In the diagram, the analog IR range sensors are of type `Evolution.RCM4AllrSensors-SharpGP2D12`. (Picture from [19]).

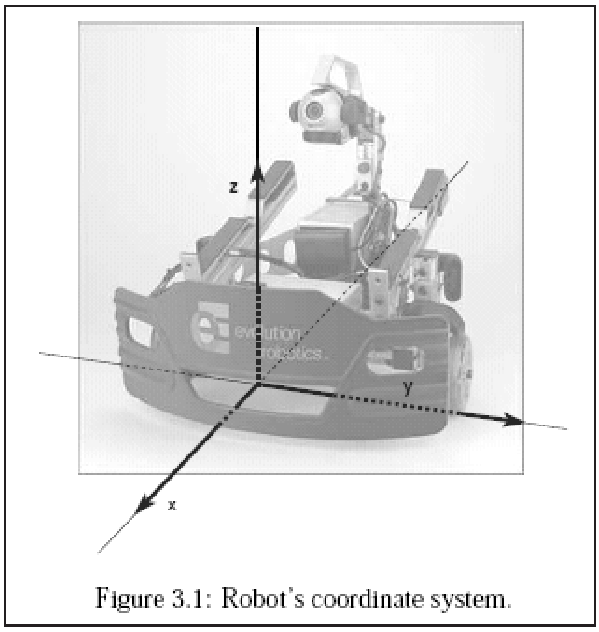

Figure 6: **The robot's coordinate system.** The robot's coordinate system as used and defined in ERSP[19]

- **Yaw:** The rotation angle around the Z-axis

A positive rotation is defined by the right-hand rule. Roll, pitch and yaw define orientation of actuators and sensors on the robots in the robot's coordinate system.

3.2.5 Units

ERSP and the Scorpion robot measures as default distance in centimeter, angle in radians and time in seconds. These units can be changed; distances to one of meters, millimeters, inches and foots; angle to degrees; time to milliseconds and hours.

4 Analysis and Design

This section has some of the most important considerations involved in the design of the ERSP Player driver. It will first look at the robots and which devices the driver can support. Then the ERSP library and use of it will be discussed. Finally, considerations about making a Player driver will be made.

4.1 Scorpion Robot Devices

The Scorpion robots have many different devices, which the driver can support. However, some of the devices are more important than others with respect to their use in robot control programs. It is therefore necessary to consider, which devices it is possible to support in Player, as well as, which devices are essential, and which devices can be left as possible extensions.

For robot control programs, the drive system, including collection of odometry data, and the IR range sensors are considered essential. Similarly, the front bumpers and the IR digital range sensor are important with respect to basic hazard detection. Combined, they provide the basic features needed for navigating the robot.

Additional possible features include mainly ERSP's power interface and the camera. Player has a power interface, which at the client side mainly allows the robot control program to check if the robot is charging. We do not find this feature very useful in the setup provided at DIKU, where all charging takes place in the same area used for storing the robots. Consequently, the power interface will not be supported. Neither will the camera interface, since we have not experimented with it at all. It may be desirable to support it in a future version of the driver in order to enable virtual Player drivers, such as the blob finder, to work.

The IBM ThinkPad laptops, on which the robot control programs run, also have several features that may be relevant to support. For example, the power state of the laptops could be exported as a Player power device. Furthermore, the harddrives comes with a Hard Drive Active Protection System (HDAPS) that can detect sudden movements and put the harddrive into a safe state to protect it from harm. Part of the system is driven by a gyroscope, from which readings can be access via the HDAPS Linux driver. It will, thus, be possible to use the HDAPS as a gyroscope. This allows the laptops either to serve as a joystick via the `playerjoy` utility program, or to use it in robot control programs when the laptop is mounted on the robot, e.g. to correct odometry data or detect inclinations.

We have focused on getting the essential devices supported, i.e. the devices for driving and acquiring odometry data, the front bumpers, as well as the analog and digital IR range sensors. Consequently, the driver will not support the power and camera interface. Since we have not been able to get the Linux HDAPS driver to work as a module, we have chosen to leave this as a possible future extension. Below, we will discuss in more detail which parts of the various interfaces will be supported.

4.2 Interfacing the ERSP Library

From the start, it has been a goal to build a driver on top of the ERSP library. Basing the driver on a software package rather than low-level hardware access involve several trade offs worth considering. By using a proven software platform for development, the driver

implementation becomes easier and developing time is cut down. This means that we will be able to have a working driver that supports most of the features available on the Scorpion robots.

On the other hand, dependencies on ERSP may also be a problem, since it adds another level of abstraction, which can be a source of problems. In the long run, it might, consequently, be desirable to use the raw serial port, to control the robot. Currently, no documentation is available from Evolution on how to do this. Therefore, it will possibly require reverse engineering the protocol for the robot control module, unless they are interested in contributing information for the development of a more low-level Player driver. However, for this project it is not relevant to look further into this option.

4.2.1 Considerations on Error Handling

It is not the job of the driver to protect the robot from harm caused by inconsiderate robot control programs. The driver should, however, ensure that the robot enters a safe state, when no robot control programs or virtual drivers are using it, or when an internal error is reported by ERSP. The former can be achieved by maintaining the number of subscribers of each device. With regard to ensuring that the robot is in a safe state, it is primarily subscriptions to the drive system that are of interest. When no subscriptions remain, the motors should be turned off.

A consequence of the Player architecture is the encapsulation and complete separation of driver code from its users. The driver has no direct contact with robot control programs, but must exclusively use the Player infrastructure for communication. This affects handling of errors reported by ERSP, where we must consider the severity of the errors and if they can be handled internally in the driver or should be propagated out to the users. Most ERSP calls we will be using return a result code that should be checked. Although all errors can potentially result in malfunction, we will first consider different types of errors, their effect, and whether some of them are recoverable.

Errors reported in the start up code in response to initialization of the ERSP core and devices should be regarded as fatal, since they are unrecoverable and most likely suggest that there is a problem in the robot control module or the wiring to it. Failure to issue motor commands should also be regarded as fatal, since this can have hazardous results, such as failing to stop the robot before it crashes into an obstacle. Although, it might be a temporary failure due to too many motor commands being issued in sequence, trying to recover will still lead to unexpected results, which means it is safest to simply error out.

Errors related to sensor readings are less fatal, since the proxy objects in the robot control program caches recent sensor readings. However, if the error persists over longer periods of time, it may eventually lead to unexpected behavior, because of the use of stale data. Although, this scenario could be handled by counting the failures and guard the result with a threshold, we have chosen to simplify the driver and always regarded these errors as fatal. The only errors that will never be fatal are odometry related failures. Mainly because we assume that robot control programs will not solely use odometry to navigate the robot and that odometry is often inexact and error-prone.

As already mentioned, fatal error should be considered unrecoverable and cause the driver to shut down and abort further driver operations. When possible, a fatal as well as non-fatal errors should output a descriptive error message to the console to make it possible to identify the cause and in the longer run, whether it is reproducible.

Player lacks a sophisticated method for propagating errors codes and messages to clients. It only allows drivers to set error codes, whereas error messages needs to be put to the console. Any error code set by the driver will cause the Player server to shutdown the driver. We will therefore only use this mechanism for reporting fatal errors. Error messaging will go through the Player provided debug print interface.

4.2.2 Mapping Player Features to ERSP

ERSP is a very comprehensive and versatile library. We will not need most of the features it provides for implementing the basic functionality of the driver. Considerations should be made about which ERSP calls are best for implementing the various Player features. Furthermore, it may be relevant to consider if any additional functionality in ERSP should be provided.

Whereas most of the sensors can be interfaced in only one way, this is not true for the drive system. ERSP provides many different methods for controlling the motors on the robot; some are very high-level, while others address different drive modes. For example, there is a method for requesting that the robot be moved forward X length units. Player supports similar requests in its `position2d` interface's `GoTo` family of methods. This suggests that a mapping can be made using the above methods. However, ERSP supports callbacks to notify about request completion, whereas Player do not. To simplify and limit the amount of state, which must be maintained by the driver, it is thus better not to use any ERSP methods depending on callbacks.

Overall, we have chosen not to use any advanced features provided by ERSP and instead focusing on using the simplest possible ERSP feature-set to control the hardware. First, they are simply not required. Second, some of ERSP's advanced features are not very transparent and they can end up getting in the way. Lastly, in the light of the desire, expressed above, to allow a future driver to use direct hardware access, limiting the dependency on ERSP will make this possible future improvement easier. As a side note, if some of the advanced features are needed, a possible extension would be to export them as virtual Player drivers.

4.3 Player Driver Considerations

In Player, there are two ways to implement and incorporate a new driver, either it can be a statically linked built-in driver or it can be a dynamically loadable plugin. For the purpose of this project, we have had as a goal to develop a stand-alone source package. This suggests making the driver a Player server plugin. Furthermore, it is much easier to develop and test a plugin driver in an external tree. Mainly because it only requires that the driver source is recompiled, since there is no extra step involving linking the Player server program. Consequently, we have chosen to develop the driver as a plugin.

ERSP is, as already described, a software package that has support for a wide range of different robot products offered by Evolution. It is therefore relevant to consider whether to write a driver specifically for the Scorpion robots or a generic driver for ERSP. Since DIKU currently only has one kind of ERSP-powered robot, it could be argued that we gain little from supporting other robots running ERSP. However, in the longer run making a generic driver is better, especially if the driver is to be included in the Player project. Finally, it could be argued that a generic driver is also easier to extend, since the resulting design will be

more modular and clean. Therefore, we have made it a goal to make the driver as generic as possible.

ERSP is a platform that has been in use for several years for developing and experimenting with robots. Consequently, a lot of ERSP-based source code already exists. When developing the ERSP Player driver, it is worthwhile to consider if some of the ERSP features that existing code depend on can be provided. This will allow the code to more easily be ported to use Player. To the best of our knowledge there does not exist any significant contributions developed for ERSP at DIKU. Since our primary goal is to develop the driver for use at DIKU and we consider this of less importance. It may, however, still be important with respect to students already familiar with ERSP, who want to port their code to run with Player. The main problem is regarding parameters that differ significantly between the interfaces provided by ERSP and Player. Some can be handled by using driver configuration settings, where as, it is not possible to find a good solution for others. We have chosen to only support this sort of portability where we feel it makes sense and where it does not compromise the cleanness of the driver implementation.

4.3.1 Device Mapping

As mentioned, the ERSP and Player device models are very different. Consequently, we must first consider which Player interfaces are most suitable for exposing the ERSP devices. Second, need to consider how best to map ERSP devices to Player devices.

Player has an `ir` interface, which are intended for IR-based range sensors. It is, therefore, a perfect fit for exposing the IR range sensors on the Scorpion robots. Player also has a `bumper` interface providing an array of bumper readings each of which can be either on or off. The front bumpers can be exposed using this interface. Player does not have a digital IR interface, however, both of the above interfaces are suitable candidates for exposing the digital IR sensors, but, since the `bumper` interface is simpler, we believe it is a better fit. Finally, the `position2d` interface can be used to export both the ERSP drive and odometry modules. However, after looking at other Player drivers we have chosen to divide up the functionality in two different devices: one dedicated to the motors and the other dedicated to odometry. By providing them as separate devices it is also likely that existing virtual drivers can more easily be used, since they tend to distinguish between the two and require them via separate devices.

With regard to mapping ERSP devices to Player devices, Player takes the approach of grouping sensors. This makes sense from the point of view that if you are interested in the readings from a sensor of a specific type you are likely also interested in others of the same type. It is our impression that this approach is superior from a programmers point of view, since it means fewer objects and less state management.

Whereas, the IR range sensors and front bumpers should unquestionably each be provided in a single device, it is less obvious how the remaining devices should be mapped from ERSP to Player. For example, dividing the digital range sensors into two devices, one for the sensors pointing up and another for the sensors pointing down could make it easier for users to distinguish between the different hazard avoidance sensors. On the other hand, this can also lead to confusion, since several similar Player devices will be available. To keep the number of device groups down, we have decided to also group all digital range sensors into a single Player device.

Finally, given that Player encourages separation of the `position2d` interface features

we have chosen to map ERSP's drive and odometry interfaces to two separate Player interfaces. Consequently, we will be able to implement them separately more easily.

4.3.2 Device Naming

Following the above design decision to group ERSP devices, we need to also address the issue of device naming. This involves both defining a naming scheme for the actual Player devices and for the individual sensors of each Player device. The former addresses the issue of making the driver able to distinguishing between identical interfaces it provides. This is necessary, since the driver will provide two `position2d` and two `bumper` interfaces. The latter allows clients to easily refer to the specific sensors, they are interested in, in a given group.

Because the choice of device names has no real effect on robot control programs–they will only be using the device number–we have chosen to simply name the devices according to what we have felt is their most visible characteristics. For example, we will use `drive` for the motor command device and `odometry` for the odometry data device. Additionally, to make the driver more generic all devices have been named, even those interfaces only provided once.

When programming the robots it should be easy to access specific sensors in a device, which consists of grouped sensors. ERSP already has a naming scheme for sensors and actuators, which is used when initializing the ERSP devices. Although, many of them are not immediately understandable, we have chosen to use the ERSP names as much as possible instead of coming up with our own naming convention. The main reasons are that the existing names may be recognizable to developers familiar with ERSP and using them makes it easier to reuse the ERSP device reference figures. To allow robot control programs to easily refer to a specific sensor the index of each sensor in a group device should be exported via a header file, which can be included in the program.

4.3.3 Configuration Settings

Player has support for specifying driver configuration options. This can be useful during driver development to conditionally enable experimental features when testing. The real purpose is, however, to make drivers more versatile with respect to different setups removing the need to recompile it to support site specific options.

ERSP does not require any configuration. E.g. it will, autonomously, figure out, which USB port should be used to connect to the robot control module on the robot. There are, however, a few parts of the driver, where considerations regarding how to translate Player features to ERSP and vice versa have not yielded a good answer. Driver settings can be used in these parts to avoid hard-coding certain values or enable multiple interchangeable solutions to conditionally be chosen at runtime. In either case we should provide sensible defaults.

One such place is with respect to ERSP's hardware avoidance system. When enabled, it will take control of the robot in certain cases and thus interfere with the expected behavior of the robot control program. Since we assume that users are aware of the possible dangers of running experiments on the robots, we have chosen to disable the avoidance system by default. However, it may be desirable to allow it to be enabled for certain situations.

Another place where options can be used is with respect to translation of motor commands from Player's `position2d` interface to ERSP. The Player interface provides a mean to set the robot's velocity and turn angle, whereas ERSP additionally, e.g. in the IDriveSystem's `move_and_turn` method, supports specification of an acceleration and an angular acceleration. Using the solutions for exercises by students on DIKU's robot experimentation course as a metrics, both of the acceleration parameters are constant in all programs we investigated. Thus, it is unlikely that not being able to change them at runtime is a problem. Furthermore, by providing them as driver settings users can set them according to their needs.

We have chosen not to implement any driver configuration settings, but instead leave it as a possible future extension.

4.3.4 Messaging

As mentioned previously, the core of Player is a queue based message passing system. Consequently, proper message handling is a very integral part of the driver. For publishing data messages containing sensor readings, most Player drivers use the main loop to first collect sensor readings. These readings are stored in the respective interface state data structure. When all have been updated the sensor readings are published to the subscribers.

A problem arises here, since a fast main loop will result in a lot of sensor data messages being published. This frequent sending causes the message queues of clients to overflow with the result of messages being dropped and thus loosing sensor readings. Normally, this is not a problem as long as the robot control program in its own feedback loop ensures that the queue is emptied often enough. However, for programs using sleep as a method of timing commands, it can result in the program dropping new sensor readings while sleeping ending up using stale data put in the queue before the sleep. Especially, if the program does not setup replace rules.

Because we will be using ERSP instead of accessing hardware directly, our main loop will be very fast. Mainly due to optimizations in ERSP, which seems to use threading to collect sensor data. We must, therefore, consider how we can limit the number of sensor data messages. Three methods will be discussed.

First, publishing of sensor updates can trivially be limited by only sending data for those Player interfaces, which have subscribers. That is, the driver will only acquire and send sensor readings from devices in which clients are interested. This can be achieved by tracking the number of subscriptions for each device.

Second, a simple rate limiting ensuring that messages are only published once every given interval can be introduced. Using this approach, the driver will hold off any new sensor measurements read between publishing intervals. Rate limiting is, however, not ideal since driver users will have to be aware of driver internals, e.g. by having to use driver settings to configure the publishing interval.

Third, limiting can take advantage of the fact that different sensors may have very different characteristics with respect to data-rate. Where readings from an IR range sensor may change frequently when the robot changes its position, the same is rarely true for a bumper sensor. The bumper sensor has only two states: bumped and not bumped, which over time will result in few state changes compared to the states reported by an IR range sensor with its more frequently changed values. These characteristics can be used to optimize messaging by only sending sensor readings when something changes. Since the client proxies mirror

the state, the robot control program will not notice this.

By only sending data on-change, the driver can end up not sending any sensor updates for long periods of time. This can, however, also be a problem, since robot control programs are supposed to periodically check and handle new messages. If the client uses blocking reads, a driver not sending any messages can consequently stall the robot control program. A non-blocking message read method exists, however, the result of such a driver will be non-portable since robot control programs not designed for this behavior will fail to work in certain cases.

We have experimented with all of the different approaches. Both where messaging of sensor updates is rate limited to once per second and where no limiting is performed. It appears to be difficult to arrive at a solution that addresses the needs of all the different programming models. The solution we have chosen is a mix of all the methods discussed above. First, we will only publish messages for devices that have subscribers. Second, we will primarily publish sensor data on-change, however, to ensure that subscribers are not starved of messages, we will use a "reverse" rate limiter to force messages to be published every second.

5 Implementation Details

Based on the design and analysis of the various aspects of the driver in the previous section, a driver has been implemented using Player C++ core library. In this section, we first present an overview of the driver in terms of the various states it transitions. We then continue to describe important aspects of the implementation along with an overview of the completeness of the driver.

The information given in this section is focused primarily on providing details about the implementation of the Player ERSP driver. It will not describe how Player drivers in general work. Information about the basics of how to write drivers are given in the tutorial for writing drivers for Player. Please refer to Appendix A to read the tutorial.

The source code for the driver can be found in the source package introduced in Section 1.6. Further information on the structure of the source package is given in Section 5.10.

5.1 Driver Overview

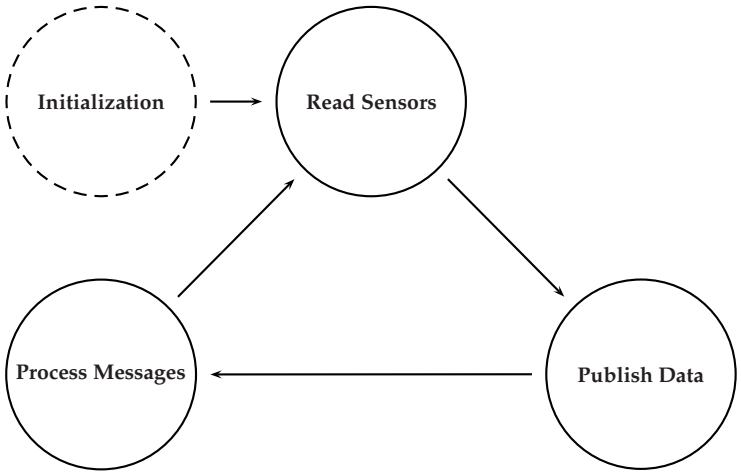

Figure 7: **The driver state graph.** The dashed circle is the entry point, after which the driver will continuously transition the remaining 3 states.

To give an overview of the driver consider the state graph of the driver in Figure 7. After initialization, the driver will enter a loop where it will first read new sensor values, publish relevant data messages, and finally process incoming messages. Concurrently with executing the loop, it will handle incoming request from clients to either subscribe or unsubscribe from devices. Each of the states will be explained in more depth below.

The initialization process has two steps. The first step happens when the Player server starts up and calls the driver's constructor. This will cause the driver to initialize its internal state and register the various devices it provides with the server. The second step is performed when the first device subscription arrives causing the driver's `Setup` function to be called. This step will setup ERSP and proceed to start a driver thread. Setup of ERSP follows the approach outlined in Section 3.2.3, where, first, a resource manager and resource container is created after which the various handles for the ERSP devices are initialized.

As shown in the state graph, the main driver loop consists of three steps. In the first step, sensor measurements and other device specific state is gathered and converted to the

units and types used by Player. All values are then stored in a set of Player interface specific datastructures ready to be published. More detail about sensor reading is given in Section 5.3, while Section 5.4 touches on some of the unit conversions taking place.

The next step in the driver main loop is to publish the new data. This is done primarily in `PubData`, which for each device checks if there are any subscribers and whether there has been any change in the device specific data. If any subscribers has been recorded and there is new device data, the data is published. Changes in device data is tracked by making a copy of all recently sent device data before it is updated by the previous step. This copy is then used to compare "now" and "before". Forcing publishing of all device data is then a simple matter of clearing the copy so all comparison will report changes. The driver uses this to force publishing every second.

The final step the driver performs is to process incoming messages. The processing will first filter messages based on their generic type into configuration requests and command messages. Each message is then further processed by matching on its subtype and device address. Where commands will simply be executed on the ERSP device handles, handling of configuration request is more subtle since it also involves sending a response. This is done by filling a message structure with the requested data and publishing it as an acknowledgement to the configuration request.

Concurrently with the above main loop, the driver must handle new requests for device subscription and unsubscription. This is done whenever the Player server calls the driver's `Subscribe` and `Unsubscribe` methods. Subscription handling consists of matching against all registered device addresses and then bumping a subscription counter up or down. For the drive device, this process also involves turning off the motors in order to put the robot into a safe state, whenever the device counter reaches zero. Issues related to concurrency with respect to subscriptions are explained in Section 5.5.

When the last subscriber and thus user of the driver has unsubscribed the Player server will automatically make the driver shutdown and release its resources by calling the driver's `Shutdown` method. This mainly consists of stopping the driver thread. One problem regarding shutdown of ERSP has forced us to keep the ERSP initialized, which means that subsequent calls to the driver's `Setup` method will not have to setup ERSP again. This problem is revisited in Section 5.9 below.

5.2 Robot and Device Description

In the analysis we decided to try and make the ERSP Player driver as generic as possible. To achieve this, we have moved most of the Scorpion specific information to a single file. The idea is to have a single place to describe the robot in terms of supported ERSP devices as well as how these devices should be grouped and mapped to Player interfaces.

For the Scorpion robot, this definition is given in the `scorpion.inc` file located in the `driver/include` directory. The robot is described via two macros: one for defining device, and another for grouping a series of defined devices into an interface. The device definition macro takes a device ID, a driver specific device type, and the ERSP device name used for initialization. The interface group macro takes the key of the device address and the name of the Player interface. Using macros defined in `ersp.inc` from the same directory, the driver can include the `scorpion.h` file to make ERSP Player driver specific types and robot device IDs available. In the driver, the `scorpion.inc` file is also used to declare a device table, which makes it easy to initialize and access devices.

Additionally, the robot definition files are implemented so the `scorpion.h` file can also be used by user programs to have access to the device IDs as well as other utility functions. Sharing the file between the driver and user programs helps to ensure that the values used are the same.

To make the driver support a different ERSP robot, a new robot specific `.inc` and `.h` file needs to be written. Furthermore, the name of the included robot definition file should be changed in the `ersp.h` file in `driver/ersp`. This is less generic than e.g. the `p2os` driver, which compiles in a table of all robot definitions, however, since we currently only have one robot to support taking that driver's approach at this time seems to be over-engineering. We have only added support for devices available on the Scorpion robot and supported by the driver. Therefore, support for additional ERSP device types may have to be added.

5.3 Reading the Sensors

One place where unit conversion takes place is when obtaining sensor values from ERSP. This process also involves converting the data types used by ERSP to those used in Player's data messages. For example, ERSP provides analog IR range distances as a `double`, while the distances for the `ir` interface in Player should be provided as a `float`. Similarly, other places in the driver related to sensor readings need to ensure that the correct data types are put into the Player data message types.

The reading of sensors must also deal with differences in thresholds between ERSP and Player/Stage. When the analog IR range sensors do not detect any obstacles, ERSP reports an infinitely big value. Stage, however, will report the maximum possible distance the sensor can measure. Consequently, we have made the IR range sensors conform to the behavior of Stage by capping any value larger than the maximum threshold of 80 cm.

5.4 Coordinate System and Units

The driver uses ERSP's global and robot coordinate system. This affects how the poses of sensors are returned in the supported configuration messages. It will also affect a future implementation of support for odometry.

As already explained, the units used by ERSP and Player are not completely compatible. The main incompatibility is that ERSP internally uses centimeters for distances, where Player uses meters. The driver therefore needs to convert all parameters involving distances, velocities, and acceleration.

Instead of the chosen solution to do the conversion manually, both Player and ERSP allows the default units to be changed. In Player, drivers must however always provide measurements in the same basic unit. Although, it would be smarter and less error prone to change ERSP to use the units of Player we have decided to leave this as a future improvement.

5.5 Concurrency and Locking

To improve the concurrency of the Player server, all Player drivers run in their own separate thread. The main entry point of the driver thread is the `Main` method of the `Driver` class, which is called after the driver has been setup. When the main loop in `Main` has been

entered, it is necessary to guard all access to variables shared with any method, which can asynchronously be called by the Player server.

The primary candidates with respect to this are the methods for subscribing and unsubscribing device clients. Since the driver will have to keep track of the number of subscriptions for each device, the main cause of race condition are the variables used for counting subscriptions. Consequently, all access to these variables must be guarded. This is achieved by putting a call to the `Lock` and `Unlock` methods of the `Driver` class before and after every access. The methods act like a semaphore and ensure exclusive access to the protected region.

5.6 Driver Build System

One of the goals of the project has been to create a source package that gives easy access to the Stage models, the ERSP Player driver, and various other helpful plugin drivers. To accomplish a portable solution that can be assured will compile on any reasonable POSIX compliant system we have felt the need to also use a portable build system for the driver. The main concern here with respect to portability is how to build dynamically loadable drivers that will work with an already installed Player server.

The build system uses GNU autoconf and GNU automake, which are also used by the Player and Stage source packages. It consists mainly of a `configure.in` file and several `Makefile.am` files, all of which are templates used by the above programs to generate a `configure` script and a `Makefile` for each directory. After bootstrapping the build system, it allows a user of the source package to first run the `configure` script to detect if and where ERSP and Player are installed, and then run `make` to build the drivers. Additionally, if ERSP is not found, no ERSP calls will be compiled in. Instead debug messages is printed to the console.

The build system is inspired by the Stage source package, which faces many of the same problems regarding the building of Player server compatible plugins. To further simplify the build process, the above build system is only used for code in the `driver` directory and the top-level `Makefile` has been equipped with rules for bootstrapping, configuring, and building the drivers. The result is that everything in the `driver` directory can be build by running `make driver-all` in the top directory of the source package.

5.7 Utilities for User and Test Programs

For testing the driver we have developed several user programs. To ease the writing of such programs, we have made several utilities, such as header files. The most noteworthy is the `scorpion.h` header file found in `driver/include`. It defines several device specific enumerated values and functions targeted for use with the Scorpion robots and the implemented ERSP Player driver.

For each device representing a group of sensors it defines indices for accessing the individual sensors. This makes it possible to refer to and use the ERSP derived sensor names. Furthermore, it defines inline functions for translating the above sensor IDs to the ERSP sensor names. This can be useful if you want to label a set of sensor values with the sensor name when dumping them to the console.

When making flexible user programs that handle command line arguments, it can often lead to redundant code when including argument parsing in every program. To simplify

this we have included an `args.h` file in the `test` directory. It is based on a similar file from the Player project, but with minor improvements. It defines a single function named `parse_args`, which should be given the arguments from to the `main` function. After being called a set of global variables will have been set according to the passed command line arguments.

5.8 Overview of Implemented Interfaces

Interface & device name	Message	Description
`position2d (drive)`	`PLAYER_POSITION2D_CMD_VEL`	Set velocity and angular velocity. Turning off motors is not supported.
	`PLAYER_POSITION2D_CMD_CAR`	Set velocity and turn angle.
`ir (range)`	`PLAYER_IR_DATA_RANGES`	Get IR range sensor data.
	`PLAYER_IR_POSE`	Get poses of the IR sensors.
`bumper (bumper)`	`PLAYER_BUMPER_DATA_STATE`	Get front bumper sensor data.
`bumper (ir)`	`PLAYER_BUMPER_DATA_STATE`	Get IR bumper sensor data.

Table 1: **Implemented interfaces and supported messages.** The first column lists the interface and the device name used in the key part of the device address. The second column list the Player interface-specific message subtype. Finally, the last column briefly describes what functionality the message provides.

An overview of supported messages for each of the implemented interfaces are given in Table 1. It only list the low-level details about what parts of the implemented interfaces are supported. Please refer to Table 4 in the user manual on page 64 for a list of supported calls. Below a small description of what functionality is not supported is given.

The `position2d` interface is very comprehensive and contain messages for controlling many aspects of the drive system that is not relevant in relation to ERSP. The most important missing parts are support for odometry data and position control–a variation to speed control–allowing clients to specify to which position the robot should move. The remaining parts, such as configuration messages for deriving the geometry of the drive system, are less relevant.

With regard to the `ir` interface, only the power configuration request is missing. The main reason is that power is not a problem for the setup at DIKU. It may, however, be relevant to support in the future as a method for turning off publishing of IR sensor data, although the C++ client library's IR proxy does not provide a method for changing this.

The two `bumper` interfaces both lack support for geometry requests. While one of the unsupported messages can be ignored, since all the bumper devices on the Scorpion robot are fixed, the other geometry message is are a possible future extension.

5.9 Known Limitations and Problems

Known limitations mostly involve features not implemented for the supported interfaces–see Table 1 to get an idea of what is implemented. Furthermore, some of the devices on the robot has specifically been chosen to be left as a possible future extension. Besides these, the driver has no known limitations.

One of the biggest problems throughout the development of the driver has been the problem of message queues overflowing. We have arrived at the conclusion that it is hard to define a way for the driver to publish sensor data, which will work with all possible user

programs. We have therefore found it necessary to constrain the sensor update rate to have reasonably working solution. Consequently, the driver uses a mix of different approaches for both limiting the number of messages as well as ensuring a steady rate of messages. This is something that should be investigated further.

Since the Player server will only have to be restarted if its configuration has been changed or one of the plugin drivers has been modified, drivers can be long-lived. This has lead to a problem since we have experienced problem with shutting down ERSP, when it is not being used. The problem is that reinitializing ERSP will lead to a segmentation fault. As a result, ERSP needs to be kept "pinned" for the remaining life of the driver once it has been setup. This means that the driver needs to manually reset some of the devices so they will appear as "unused" when a client resubscribes. For example, a possible future version supporting odometry will have to reset the odometry device so old data from previous usage is not reported.

With respect to error handling, it has been difficult to find a good solution for handling the problem of the USB cable between the laptop and the robots control module being disconnected. Whenever this happens ERSP will start to continuously write warning messages to the console. Stopping the current user program does not cause warnings to disappear. We believe this is mainly because the ERSP handles, as mentioned above, due to problems remain initialized, when no clients are using the driver. To workaround this problem, it is necessary to force the Player server to shutdown.

During testing we have experience a problem with the readings returned by the analog IR range sensors. As mentioned in the above section on sensor readings, the driver applies several conversions on the sensor values before they are published. However, we believe none of the conversions can explain the inconsistencies of reported values. Furthermore, printing the values unmodified from the driver paints the same picture. We will investigate this further in the evaluation.

5.10 The Software Package

This section outlines the content of the source package introduced in Section 1.6. Refer to that section for instructions on how to obtain the source package. Below a few details about the source package directory structure and content is given.

Assuming the source package have been checked out from the repository you should have a directory named `erspplayerdriver` which is the source package. It has the following subdirectories:

`driver`: Contains all the driver related files.

`player`: Contains configuration files for the Player server

`stage`: Contains all stage configuration files separated in subdirectories of the world they simulate.

`test` and `ersp-test`: Both contain robot control program used as examples and for testing.

To get you started, read the `README`-file in the `erspplayerdriver` directory.

6 Stage Modelling

To use the Stage 2D simulator when prototyping or developing robot control programs we have to model the Scorpion robot and create a simulated environment for the robot to move around in. Both the robot and the environment with other objects have to be modelled and described for Stage so it can be simulated as precise as possible compared to the real world.

This section summarizes our consideration and work with creating the robot model and a model of a limited environment of a real world. We will not go into details about Stage and how to configure and use it, but focus on essential details that impact our possibilities and choices for the model.

6.1 Model Constraints in Stage

A simulator compared to a real world will typically have many constraints on how realistic the modelling of the real world can be. This of course applies to Stage modelling as well. First off, Stage is limited to modelling two dimensional worlds. Furthermore, the authors states: "Stage is designed to support research into multi-agent autonomous systems, so it provides fairly simple, computationally cheap models of lots of devices rather than attempting to emulate any device with great fidelity." [6]

Another limitation is the selection of interfaces and models that can be used to describe the robot and other objects in the environment. Stage can of course only supports a limited number of such interfaces and models.

6.2 Models and Interfaces

Stage supports several sensors and actuator models, like sonar sensor, scanning laser range finder, mobile robot base with odometry or global localization. Stage simulates a collection of models specified as a world. In this world, models can be used to "build" new models. Typically one of the models will be a robot, but some of the models are used for creating other object, such as the underlying map (e.g. a floorplan) or obstacles. There are some special models that are used for general purposes like specifying properties of the world (the simulated environment). The GUI that Stage presents is also an object that has properties, such as size, and has to be part of the Stage world.

Since Stage is a driver to Player, Stage implements and supports interfaces that enables the user program to control the robot. The robot is a model in Stage that is defined from other models. A Scorpion robot could be composed of a mobile robot base with odometry model, a IR range model and bumper model. Stage therefore has to provide an interface to support each model, just like a driver that supports each piece of hardware on a robot. In this example, Stage could support the `position2d` interface for the mobile robot base, the `ir` interface for the IR range model and the `bumper` interface for the used bumper model.

Below we will outline our model of the Scorpion robot and how a realistic real world environment is modelled.

6.3 Scorpion Robot Model

We want to model the Scorpion robot as precise as possible, so we will be able to work in a simulated environment with as little differences to the real world as possible. Consequently,

we want to model all actuators and sensors and describe their position and orientation. In addition to this, we have to model the actuators behavior and the sensors functionality.

Using the Scorpion robot hardware as a basis, we have to model the driving system that moves the robot, the 2 types of IR sensors, the bumper, and physical aspects of the robot like size, orientation, etc.

Stage configuration files that models environment and the Scorpion robot is found in the software package described in Section 1.6. They are well commented and can serve as examples for further modelling and writing of Stage configuration files.

6.3.1 Physical Aspects

It is possible to set a position, size, and a center of a modelled object in Stage. As Stage is a 2D simulator, we will model the robots size as its bounding box projected vertical onto the floor plan. Its dimensions are given in Figure 3 on page 18 and can also be found in the *ERSP resource configuration file* as `Dimensions` [19].

The robot's dimensions are given in centimeters in the documentation. Player/Stage configuration is done in meters, so we round to nearest centimeter. The robot's center coincides with origo of the robot's coordinate system and is therefore a negative x-value with respect to the front of the robot. The robot's orientation is straight forward with an angle of zero.

Beside the physical aspects of the model in Stage, the model has to be visualized. This is done with simple lines and polygon drawing. The robot is visualized as its bounding box (rectangle). The robot's form will appear as a rectangle with the sides almost being the same length, which is a simplification of the real robot. The real robot has rounded edges in front because of the bumper and does not fill out the entire space in the bounding box on each side of the support wheel in the back.

Driving System

The Scorpion robot has a differential steering model as it has a separate motor for each wheel. Its 3rd support wheel at the back does not affect the steering model.

The robot's odometry and feedback from these could be chosen to be precise in the simulator model or be chosen with erroneous measures, which would be more realistic. We have not calibrated and analysed the errors in the odometry on the real robot. Together with the expectation of Stage being used for prototyping, we have chosen a precise odometry model for easier testing of simple programs that does do error correction for odometry.

IR Analog Range Sensors

Modelling the IR analog range sensors in Stage is done by specifying each sensors size, position, angle. The range and view angle of the sensors are also easily specified. Thus, modelling the range sensors in Stage should be easy. Using the sensors in Stage, though, is not quite supported as Stage does not support the `ir` interface. Stage has support for the `sonar` interface, that the documentation states can be used instead of an IR range finder. This means that programs using Stage will have to use the IR range sensors as sonar device interface. This makes it impossible for user programs to be ported between Player and Stage without modifications. To workaround this issue, we have created a simple driver for mapping the `sonar` interface that Stage supports to an `ir` interface. The driver, available in

the `driver/stage` directory of the source package, is called `sonar2ir` and supports the minimum amount of features necessary to have a working `ir` interface in the simulator.

Furthermore, it should be noted that there is a little technicality about defining the sensors in Stage. If all sensor were identical size, the sensors range and view angle could be defined once in an array and would apply to all sensors. The fact that the robot has two sensors that are mounted vertical, makes it necessary use the alternative way to specify these properties. Normally, this would mean we have to specify the size, position and angle separately for these two sensors. Individually specifying these properties seems though not to work in Stage. We therefore had to define all sensors with the same size and this means that the two vertically mounted sensors are not visualized correctly. We do not expect it to impact the simulation results, but would like to note that the robot shows these sensor horizontal mounted, which they are not.

Another implementation detail is that the specification of the sensors should occur in the same order as in the driver or vice-versa. Player bundles sensors in array and the index of each sensor in Stage is assigned in the order they are listed in the configuration file. The first sensor in the Stage configuration file is the first in the array when using it in the user program. This is important to remember as it easily could lead to strange results in the user programs, unless the user program asks for the pose of the sensors and uses them to select the correct sensor for the current computation.

As a final remark, it is not documented, whether the IR range sensor model in Stage uses the view angle only for visualizing the sensor, or whether it impacts range measurements. We believe to have experienced through our experiments that Stage actually only measure distances from range sensor as a traceline from the sensor and not within the area of the visualized viewed angle and spread beam. As we have not found any facts about the view angle of the IR range sensor in the documentation for the sensor, we have chosen to set the view angle to 5 degrees spread as the maximum range. This probably is too much, but because Stage uses a trace, it does not impact the simulation and a 5 degree view angle makes the visualization more clear.

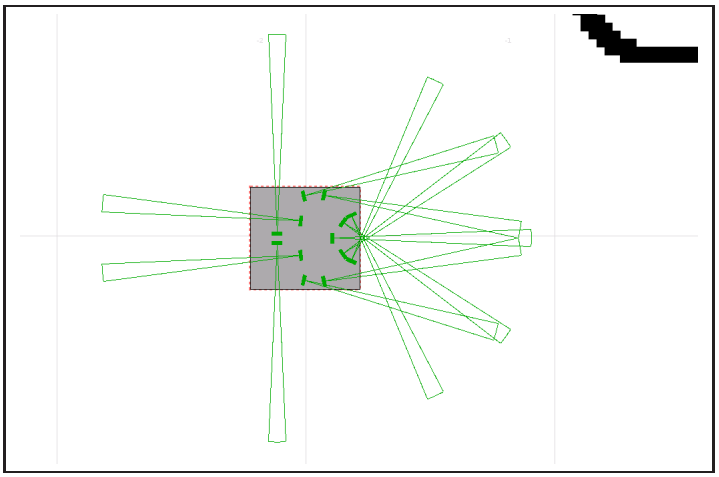

Figure 8: **The sensor model in Stage.** Stage showing the modelled sensor of the Scorpion robot.

In Figure 8, a picture from the Stage simulator of the robot is visualized with the IR range sensors.

IR Digital Range Sensors

The digital range sensors can not be modelled in Stage because of two reasons. First of all, the digital range sensors on the Scorpion robot are pointing up and down for detecting ledges and hangout to avoid hazardous fall-downs or damage to the camera. Such detections do not make sense in a planar 2D world. Second, there is no Stage model for the digital range sensor, though the model of the analog IR sensor could be used as a replacement, if the user program did some thresholding on the readings according to the specification of the real sensor.

Consequently, the Scorpion robot model in Stage have no IR digital range sensors nor does it support a device interface for such sensors.

Bumper

As with the digital range sensors, Stage have no model for bumpers and does not support a device interface for bumpers. Therefore they cannot be modelled in Stage. The fact that the bumpers cannot be modelled will make the Stage model less accurate. Worse is, if the user program uses bumpers, it will not run under Stage without modifications.

The Stage version before 2.0.x supported the bumper device interface, but it has not been ported to this version yet.

6.4 Real World Environment Model

Real world environment modelling means to build a Stage model of some well-defined an limited part of the real world. That could be a floor plan or some small area of a floor.

Stage has general model that can be used for such purposes. Based on a bitmap picture, elements in the picture, like lines or boxes, can be defined as uncrossable for the robot. E.g. the robot can drive around in a bitmap of a floor plan with white background and walls drawn with black lines. The black lines will act as wall that robot can detect and interact with, e.g. drive into and force the robot to stop. Such maps are typically specified with a bounding box that makes the frame of the map uncrossable as well.

We will not go into details on how to define such a world as there are several examples of defined Stage models among the Stage configuration files in the software package[3].

We will continue the rest of this section by describing how to make the modelled world match the part of the real world that we wish to model. Consider the following case, where we want to write a user program that can navigate the robot to follow a wall in the real world. We have build such a wall in the real world on the floor in our robot lab and want to model the area in Stage to let us use the simulator for prototyping. The prototype of the program will hopefully act the same way in the real world, if the modelled world is realistic and the robot has been modelled accurately.

In a square area of 5 meters, we have placed a wall of masonite plates along two sides of the square such that the walls meet in a corner. The plates define our wall which is 3.6 m long on one side and 2.4 m long and the other side. How can this be modelled in Stage?

First of all, we have to create the bitmap picture that illustrate the wall. We call this wall *dikuwall*, which is also the name of the Stage example and partly the name of the user program: `dikuwallfollow`. The picture should be a white background with the wall drawn

[3] see Section 1.6

with lines. Stage supports scaling of both the world and the underlying bitmap, so the most important thing when drawing the bitmap is to keep the interval scales correct between the elements in the drawing. The size of the picture's bounding box should also be known in real world sizes as this is used for setting the correct scale in Stage.

The resulting bitmap, we created, was drawn in a vector drawing program that uses scales and coordinates. We drew the picture in a scale of $1:10$, making the 2.4 m wall exactly 24 cm long on the bitmap. The bounding box of the picture is in the drawing program's measure set to 50 cm, as this gives 5 meters in scale $1:10$.

Finally, the Stage model was defined with properties of a typical map model and the size of the model was set to $5x5$ meters. The size of the Stage world was set to the same, but could have been defined larger.

In the evaluation given in Section 7, we will evaluate the use of this world as a real world model and how the prototype user program worked in the real world afterwards. As for now, the model can be seen in Figure 9.

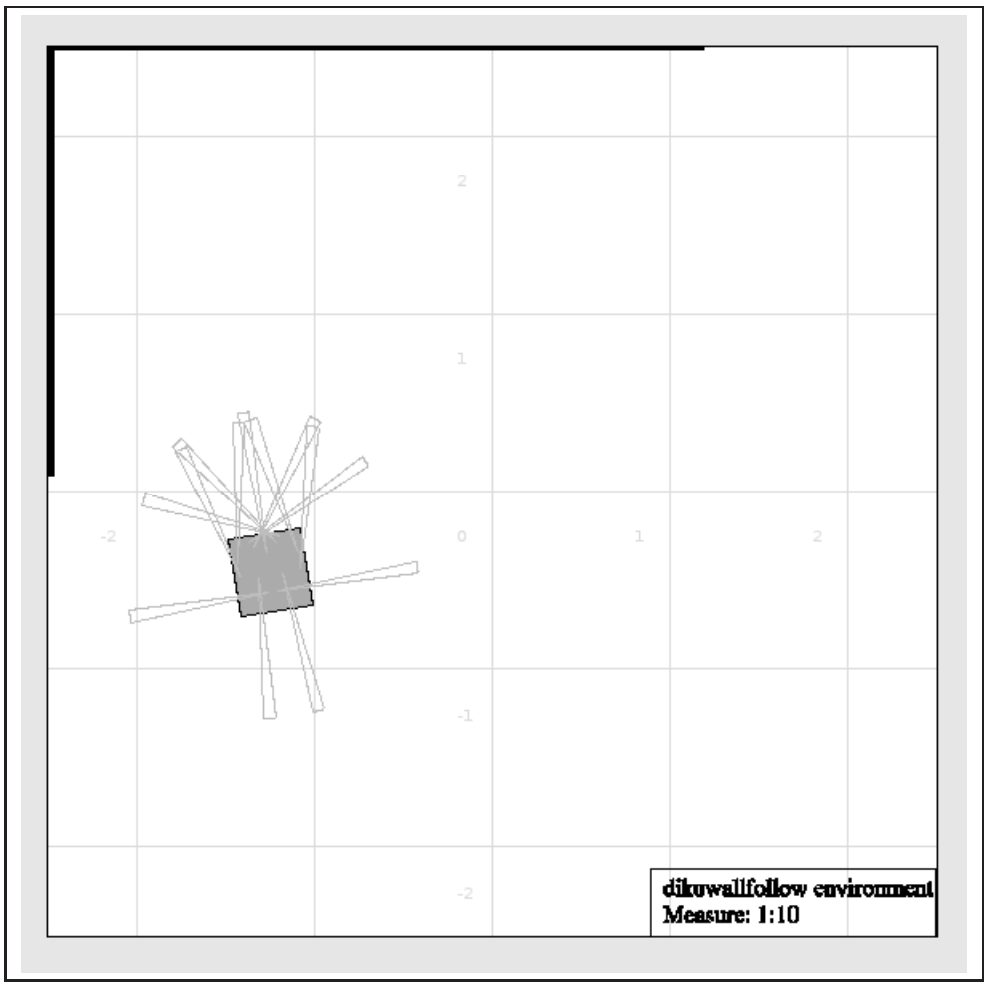

Figure 9: **The dikuwall real world environment model** This screenshot from the Stage simulator shows the modelled world of the 5 meter floor area with the masonite plate wall in the upper left corner.

7 Test and Evaluation

As a summary of this project we will illustrate the implemented driver and its functionality and give at short test and evaluation of the driver. We will also evaluate the use of Player/Stage through some experiments and highlight some of the new possibilities the Player framework provides.

All our programs that will be used or referenced in the experiments reside in our source package, as they besides testing our driver are being used as examples and tutorials. Video material that documents and illustrate the evaluation and our findings are also available in the *video archive*. See Section 1.6 on how to obtain the source package and gain access to the *video archive*.

7.1 Driver Test

The implemented ERSP Player driver was tested and evaluated during the development. We have found the driver to work as expected regarding our primary objective for the implementation. The driver implements driving interface and sensor reading possibilities and they have been used in robot control programs.

Below we will look further into the driver's capabilities and the use of the hardware support the driver implements. It will not be a thorough test, but merely an illustration of result and finding we think are important to notice or explain.

7.1.1 Driving Experiment

We will do a simple experiment to illustrate the implemented driving interface can drive the robot. The experiment is summarized in Table 2.

Results and finding: In the videos of the two square programs, we observe the robot as expected drives in a counter-clockwise square. There is a little difference between the Player and the ERSP version of the program, as the Player version does not stop the robot at the same spot as the ERSP version does. The difference occurs because the program flow is not identical and the Player program stops turning a little later than the ERSP program. The random walk program shows the robot can drive in other directions and with other turn rates. The random walk is also used in the sensor experiment below where we will comment more on the sensor use in the program. Thus we have illustrated with this simple experiment that the Player framework can be used to drive the robot.

7.1.2 Sensors

This experiment should illustrate the implemented sensors used in Player. The experiment is summarized in Table 3.

Results and finding: First a description of how the experiment was done, as results are not video-recorded. As the video-clip shows, we activate the IR range sensors using a solid piece of masonite. It is put in place in front of some of the sensors for a few seconds and then moved to the next sensors. This should let the sensor reading shows the distance to the object. The bump sensor is activated with the food. The IR digital sensor, detecting ledge and hang hazard is activated respectively by lifting the robot and moving a hand over the hang-out detection sensors.

Sensor readings (`output`) The results are not as expected. We see one problem. The sensor reading are not stable, even though they are activated in a way so there should be time enough for the sensor to show stable reading. The IR analog sensors show readings that indicate the object and at a distance they sometimes seem correct. But the sensor reading oscillate, both 5 and 15 percent, where they should be stable. The IR digital ledge and overhang sensors show same problem as they change between activated and not activated.

Bump sensor reading The bump sensor readings seem to be stable and react as expected on activation using the `output` program. The experiment with `bumper` program is also as expected, as activating the bump sensor immediately makes the robot drive as programmed.

Usage of sensors in the `randomwalk` program This program drives the robot quite as expected, even though it too, must be affected by unstable sensor readings. It should be noticed that the obstacle avoid-phase of the program is done the moment just one sensor reads less than the maximum distance. Therefore unstable reading will have less effect on this program.

There are two possible explanations for unstable sensor readings. First, it could be the sensors that themselves are not stable and oscillates because of disturbance, such as changing light conditions or the surface of the masonite we tested with. Second, it could be a problem in the Player driver we have implemented - possible some problems with messages and queues. The first explanation is easy to examine and decide if this is the case. This could be done by writing a program that works only with ERSP acting like the `output` program and under the same conditions. This way we can eliminate if the Player driver affect the readings.

We will get back to this finding and investigate the problem further. This is done in Section 7.1.3 below, where we do some further testing.

7.1.3 Sensor Reading and Update Problem

This section is dedicated to investigate the found problem of unstable sensors in the sensor experiment above.

Our strategy will be to evaluate the sensors' stability with a program written for Player and one written using only ERSP. That way we can decide if the problem of oscillating sensors only occur in the Player program and thus probably is caused by the driver or Player framework. It the sensors are unstable in ERSP the earlier finding is just caused by noise from e.g. disturbing light.

We have rewritten the `output` program to a new program called `output2col` which print out analog IR sensor readings in columns to the terminal for easy capturing to files for later analysis. The new program uses only analog IR sensor, as we expect the cause of the problem to be same for the two sensor types. We have made a comparable program that uses ERSP called `ersp-output2col`.

We intend to compare results from running the programs on the same robot with the exact same setup and some obstacles placed around the robot.

Driving experiment	
Program(s):	`square`, `square_ERSP`, `randomwalk`
Purpose:	The experiment is done to show that the implemented driving interface in Player work and can drive the robot.
Description:	We will use the robot control program `square` for Player and the `square_ersp` for ERSP to illustrate this and see if there are any differences on the two platforms. The program is very simple and will make the robot go in a square and this will therefore merely be an illustration of the driving and not a thorough test. We extend the experiment using the Player robot control program `randomwalk`, which is a more complex program using a read-think-act loop and driving random around in the environment. It uses sensors, that will be tested below.
Expectation:	We expect to see the two square programs drive the robot in a square counter-clockwise. There should be no or little difference between the two square programs. The random walk program should just illustrate the robot driving random around and showing that it can drive/turn in different directions.
Material:	We have recorded videos and pictures of the robot running the programs and they are available in the *video archive* as explained in Section 1.6. The video and pictures are named accordingly to the program that drives the robot and placed in the subfolder `drivingexperiment`. E.g. `square` program is illustrated in video `playertest-square.avi` in the `drivingexperiment` folder in the *video archive*.

Table 2: **Driving experiment description**

Sensor experiment	
Program(s):	`output`, `bumper`, `randomwalk`
Purpose:	We will show that the implemented sensors on the robot are available in Player through the `ir` and `bumper` interfaces.
Description:	The `bumper` program uses the bumper interface and will react on bumper input which we will show. The bumper input makes the robot go backward and turn in the backward direction of the side it was bumped. The output program uses all sensor interfaces and will show sensor reading and bumper sensor activation. It will also illustrate the the digital sensors working with the `ir::bumper` interface. The random walk program will show use of sensors in work in an real environment as the program will make the robot drive around randomly avoiding obstacles.
Expectations:	Using the `output` program we expect to see sensor readings change according to the activation of sensors on the robot. The effect should show immediately and consistently. The random walk program should make the robot drive safely around in an area with obstacles avoiding them. It relies only on the IR range sensor which can be seen as an evaluation of the use of these in an real experiment. Of course we expect to see the robot not hit any obstacles, as the safety distance is the maximal sensor measure distance of 0.8 meters.
Material:	A small video is available showing the robot reacting on the bumper running the `bumper` program. As `output` writes output to the standard out on the terminal we can't illustrate with a video and the idea of a screen-captured video did not succeed within a reasonable amount of time. We therefore will tend to describe how the experiment was done and the output. A small video-clip of how the robots sensor was activated is however available. The video of the random walk program in action is found in the subfolder of the previous experiment described in Table 2 where as the other videos of this experiment is found in a subfolder named `sensorexperiment` also in the *video archive*.

Table 3: **Sensor experiment description**

In Figure 10, two graphs show the sensor values from the output of the 2 program. Sensors are approximately read once every second and was "recorded" in an experiment where the robot was parked and surrounded by objects to activate the sensors.

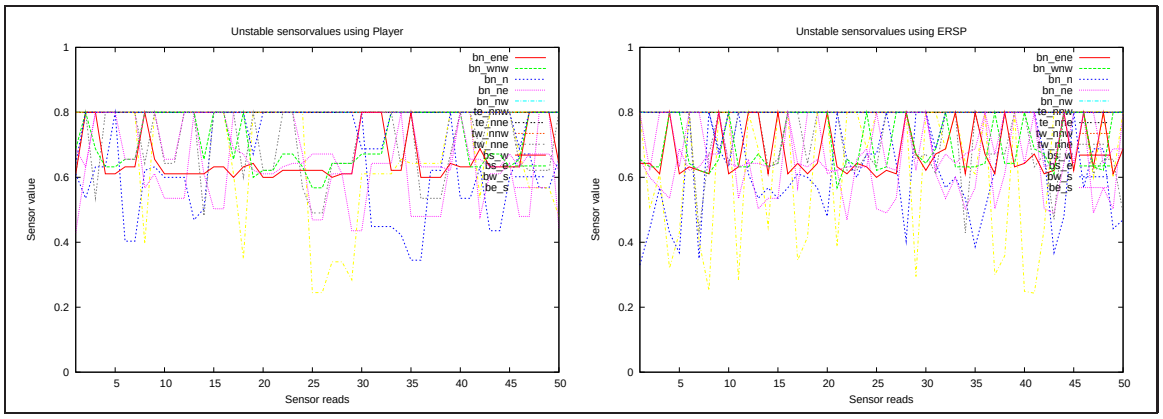

Figure 10: **Unstable IR sensor readings.** The 2 graphs show IR sensor range values as a function on the number of time the sensor is read. The graph shows the readings are very unstable in both under ERSP (right) and Player (left).

The 2 graphs show the different IR sensors value for each time they are read (only the first 50 readings are shown). We see that the values are highly unstable in both ERSP and Player. There are few values that are stable and show maximum value because the sensors were not activated.

We therefore conclude that the unstable readings of the sensors, both IR analog and digital, do not result from the implemented Player driver as they also exist when using the ERSP software directly. The oscillation of the values, therefore, probably is noise from light source in the surroundings. It could be further investigated, which error function could model the noise to correct for the noise in more advanced programs. This, however, is not something we will look further into in this project.

Not shown in then graph, but investigated, was sensor readings in Stage. They showed as expected to be fully stable. Stage support including an error function when modelling sensors that could be investigated further. It could be an idea to try to model the error from the real sensors in Stage.

7.1.4 Summary

Our evaluation of the implemented driver shows in the above experiments it can be used for writing a robot control program that use the Scorpion robots sensors and actuators. The finding of the unstable sensors shows a problem is passed from ERSP to Player and therefore the Player driver as well as ERSP could be used regarding this challenge.

We therefore conclude that our primary goal of writing a driver for the Scorpion robot based on ERSP is fulfilled. There of course may still be limitations and considerations that show up in the further evaluation in the sections below.

7.2 Stage Modelling Evaluation

We will in the following try to evaluate the modelling done in Stage as described in Section 6. This evaluation includes using Stage as a simulator for writing robot control program that later with as few changes as possibles should work on a real robot.

7.2.1 Robot Model

We first look at the robot model created and shown in Figure 11.

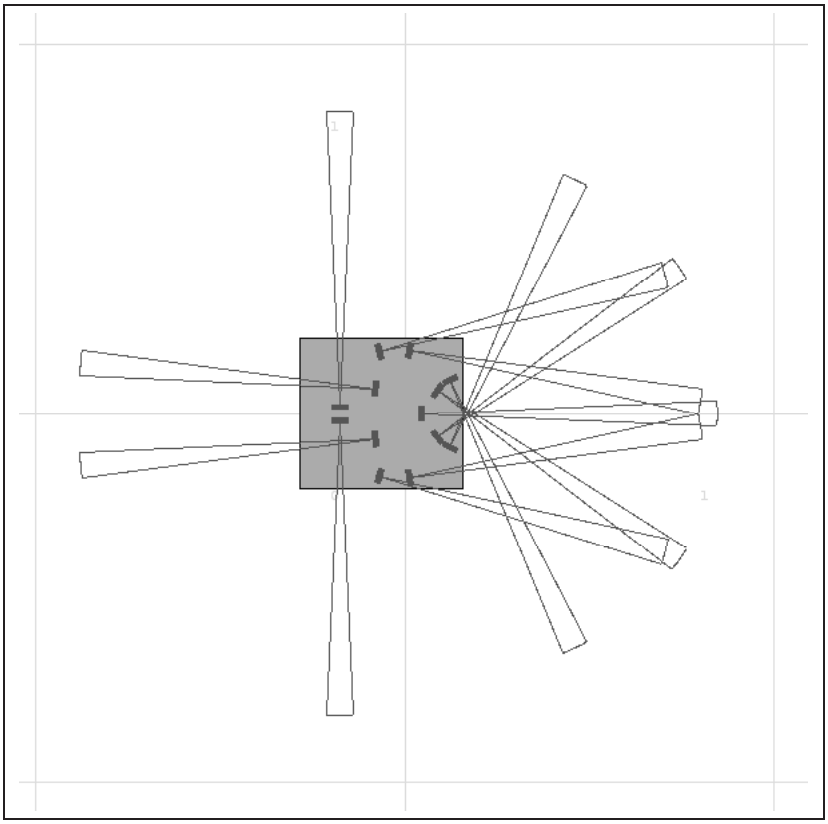

Figure 11: **The Scorpion robot model used in Stage.** The screenshot from Stage shows a zoomed view of the robot with its IR range sensors illustrated (*ranger data* enabled under view in Stage). The robot is positioned with its center exactly in the center of the map that shows quadrants with measures 1×1 meters.

The figure shows the robot as a simple almost-square (is modelled as 44×40 cm). Not the most realistic representation, but the expected representation as we defined it. Doing simple measure on the figure we can see the robot has approximately the dimension we wanted. We also see the sensors spans approximately 80 cm as each of the 4 squares is 1×1 meter.

The visualization of the robot could be done better regarding its shape as the not all correct shape as a square could influence the simulations. The problem is the shape is not a quite as a square in the rear section of the robot. The form is more triangular in the rear section because of the 3rd support wheel.

In Figure 12, we see that if the robot is very close to an obstacle the square will as shown on the picture prevent the robot in turning around its own axis. The corner of the square

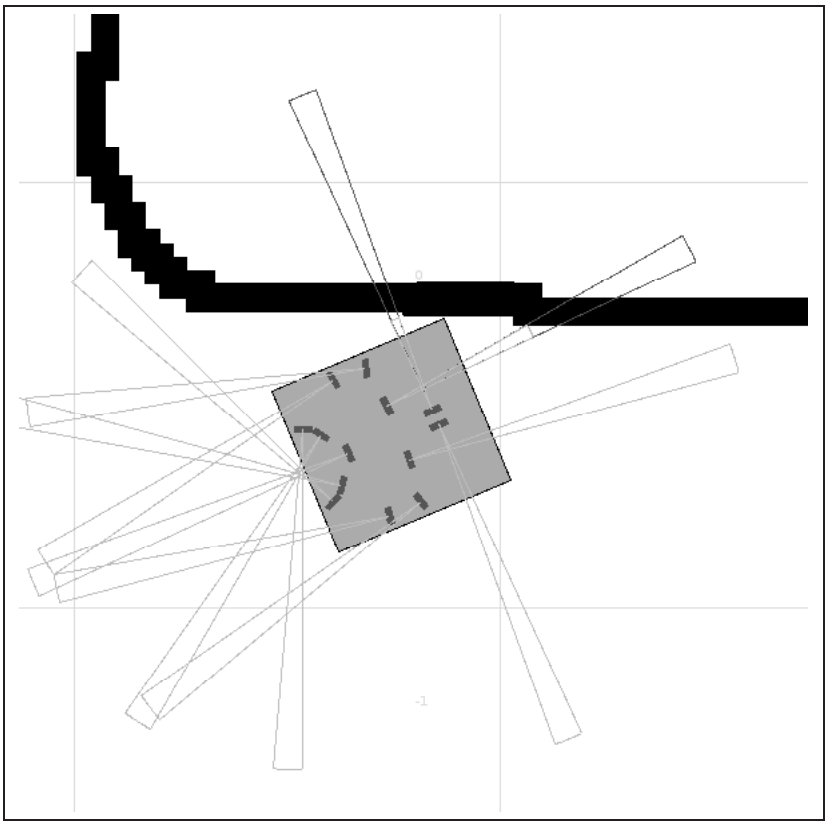

Figure 12: **The Scorpion robot model illustrating a problem.** A corner of the square representing the robot has hit the obstacles and the robot is stock, as it only tries to turn around its own axis. This of course is not a problem if it goes forward at the same time, but often only turning near obstacles is done. A more triangular form that better represent the support wheel in the rear section would probably prevent this problem.

hits the obstacle and the robot is stock. If the robot was more triangular in the rear section it would probably not get stock.

An extension of the robot model could also be adding "eye candy" like wheels and the bumper could be drawn even though it is not used.

Figure 13 shows another problem regarding the model of the sensors of the robot. The sensor visualization seems to match the actual position on the real robot really well, but the question is as earlier stated the angle for the sensor readings that is modelled as 5 degrees. This probably is too much, but was kept as it was regarded not to have any effect on the simulation in Stage. This is, as the figure shows, not quite true, as the sensor with a spreading of 5 degrees activate on the obstacles before the sensor defined with 1 degree spread. Thus it probably would be more safe to use 1 degree spread for the IR sensors, but still we have found no official documentation for the sensors on the actual values.

Even though we have shown some problem in the model of the robot for Stage, we believe it has no real effect in a simulation. The simple model as used above will give as good results as a more advance model. Most robot control programs will try to avoid obstacles as early as possible and this make the robot keep distances to obstacles where the square shape is not a problem. The sensor choice of 5 degrees has no real effect either, when running a

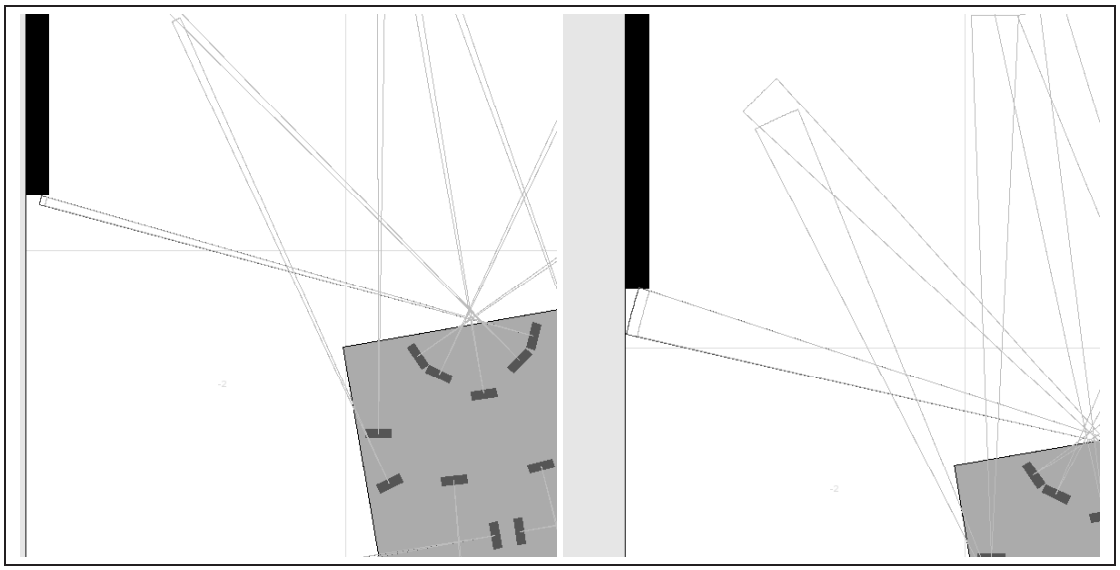

Figure 13: **The Scorpion robot model illustrating the effect of the spread of IR sensors.** The figures show that the sensor model with a spread of 5 degrees activates before the one with 1 degree of spread. This is contrary to what we earlier believed, but there is still found no official documentation for the actual sensor specification.

simulation of control program.

Figure 14 and 15 show examples and explain why these model simplification has no real effect in the simulations.

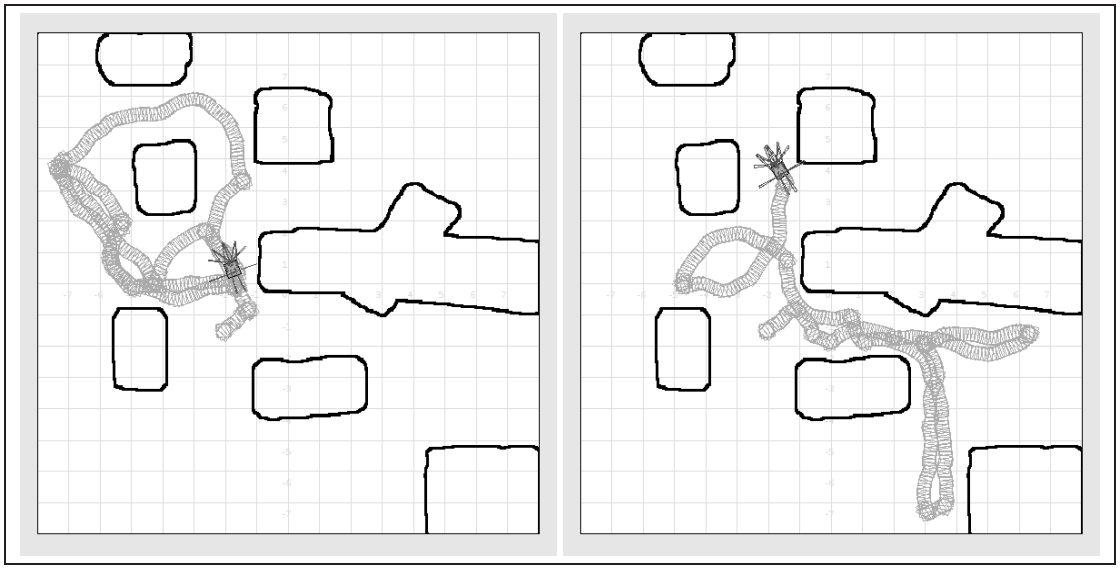

Figure 14: **Two alike simulations done with different angle spread for IR sensors** The simulation on the left is done with the robot model having 1 degree angle spread for the IR sensor model and the simulation on the right with 5 degrees. The robots movement is traced in the environment and as mentioned the robots safely avoid obstacles with both sensor models.

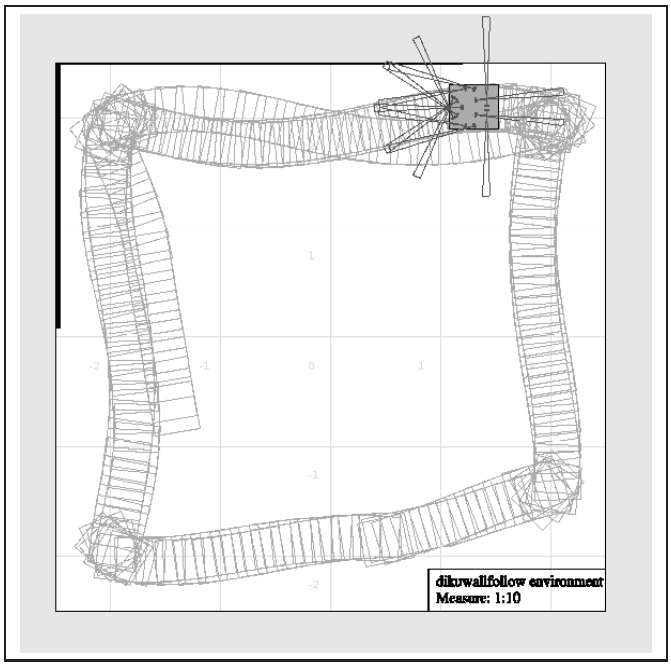

Figure 15: **Robot doing a wall follow** The pictures shows the robot tracing a path of its wall follow control program. The wall follow program is done as safe as possible and therefore the square shape of the robot will not pass close enough to the wall to hit and stop the robot. It is our understanding that one always will try to keep the robot as long away from obstacles as possible, which means that the simple model of the robot not necessarily have to be improved.

7.2.2 Experiments in a Real World Model

In this section, we will evaluate how well Stage can be used to model a real world environment and a real robot. The purpose is to find out how well Stage can be used to develop a robot control program that, without modification, can be moved to the real robot and used in the modelled real environment without noticing any differences.

It could be a goal if using a simulator, to do most of the development in the simulator and finally move to the real robot and environment without have to do any modification of the program except very small adjustments of for example few parameters.

Our approach will be to conduct an experiment with the previously created real world model and the robot model above. The experiment is done running a program written only using the simulator and the two models. Finally, trying to move the the modelled real world environment and robot and evaluate our findings.

The real world environment and model is described in Section 6.4 and showed in Figure 9. How well a model of the real world is, will be hard to evaluate without using it in a simulation.

The program that will be used is the `dikuwallfollow` program and the Stage world and configuration is named `dikuwall`. All files are found in our source package. The real world experiment is documented with video-clips that are available in our video archive.

The program we will use in the experiment is a wall follow program and an example of

how the program make the robot follow a wall is illustrated in Figure 16. The world in this figure was created only to illustrate the program in a larger environment than the one we are going to evaluate in a moment.

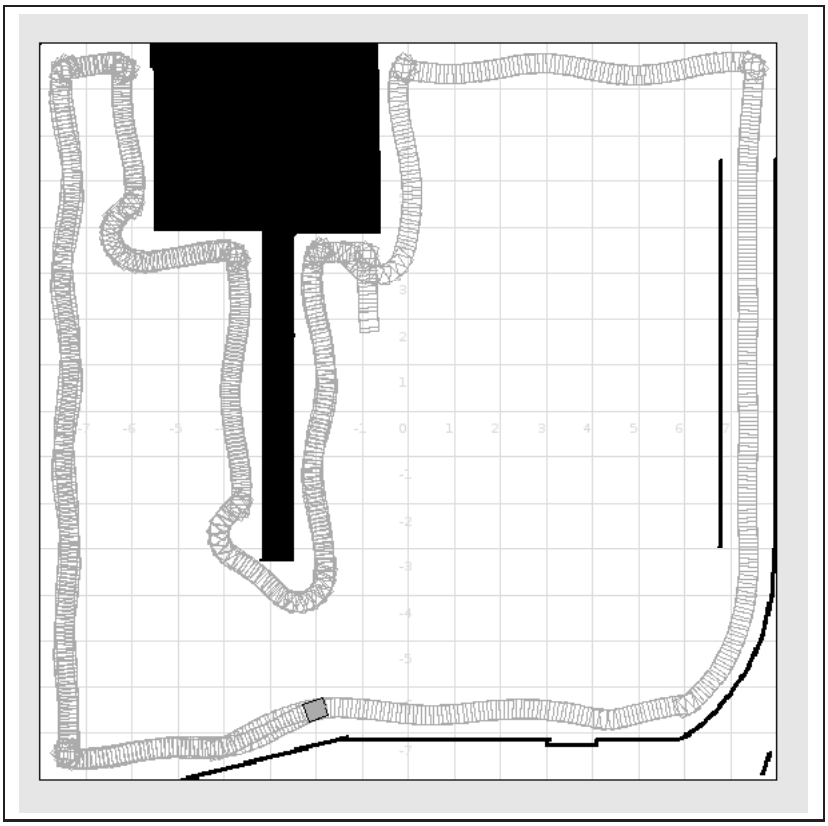

Figure 16: **A large environment simulated in Stage.** The robot is controlled by the `dikuwallfollow` program that makes the robot follow the wall it first encounters. It it the program we in a moment will evaluate in small simulated environment and thereafter in the same environment in the real world. The figure here illustrates the robustness of the wall follow program in the simulator as the robots path has been traced in the picture.

We will now show a series of figures, that illustrate 5 experiments with the `dikuwallfollow` program run under Player/Stage in the simulated environment and afterward under Player in the real world environment. The videos from the real world experiments are found in the video archive under the subfolder `dikuwallfollow`. We can reveal now that *all* the real world experiments with the program did not go as planned. The robot "crashes" the wall or spins around itself change direction to follow the wall. Therefore we have marked on the figures where the real world experiment went wrong. we will try to explain the finding below.

In Figure 17 and 18, the experiments and illustrated traces of the robots path all show that the program work in the simulator but not the real world. Please refer the video archive for small video-clip of the experiment.

There could be 2 reasons that can explain our finding. The first could be the outcome of what we tries to evaluate, that is if the change between Stage and the real world is possible. We do not believe this, as we will get back to in a moment. The second reason could be our

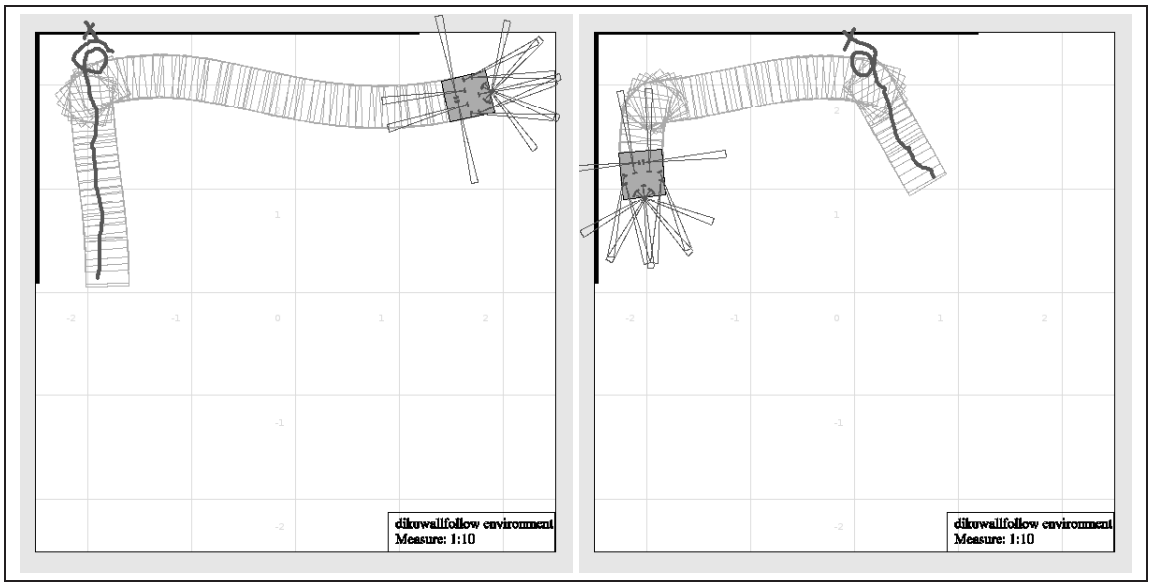

Figure 17: **Illustration of the dikuwallfollow experiment run 1 and 2** The gray box-trace shows the experiment run in Player/Stage and thin hand drawn line trace is an illustration of the robots behavior in the video clip in the video archive. The robots behaves as expected in the simulator but crashes the wall in the real world, when the program in run under Player on the Scorpion robot.

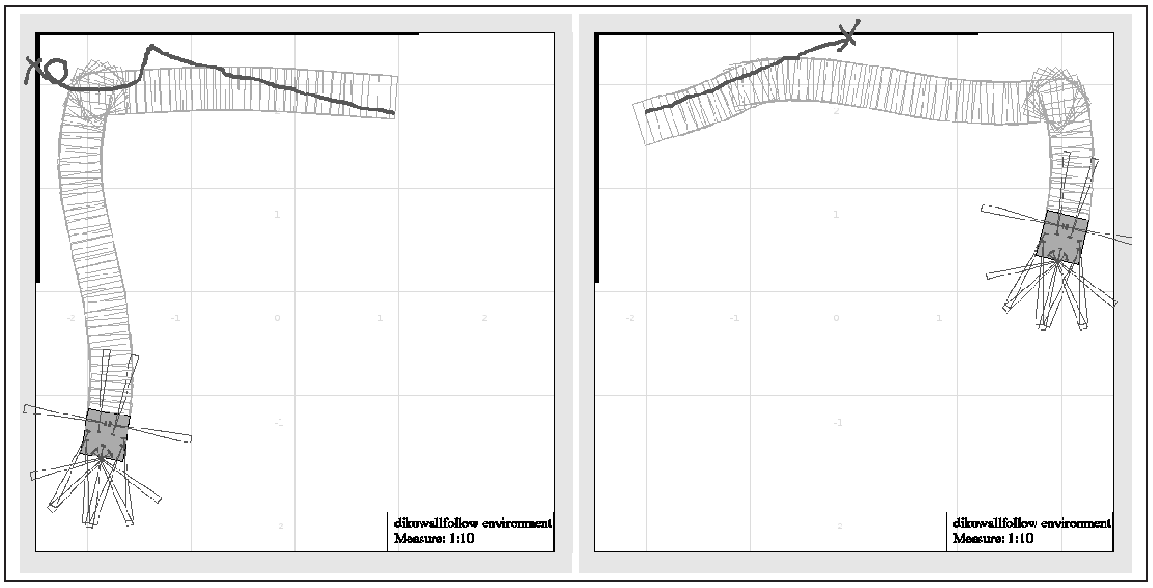

Figure 18: **Illustration of the dikuwallfollow experiment run 3 and 4** Again in these to runs we see that the robots behaves as expected in the simulator but crashes the wall in the real world, when the program in run under Player on the Scorpion robot. (The gray box-trace shows the experiment run in Player/Stage and thin hand drawn line trace is an illustration of the robots behavior in the video clip in the video archive.)

implemented driver not working as expected.

The first reason does not seem feasible, since the simulator is widely used for prototyping and we have earlier in this project used with programs that worked well in both worlds. We of course would expect small differences when moving from an ideal simulated world to

the real world with noise and other factors. But the differences should not have been that dramatic making the robot navigate into the wall.

Therefore, the problems point in direction of our driver as the second and possible explanation. We also arrive at this conclusion when we investigate the video-clips into details. If we look at the video-clip of run 2, where we have cut strategic frames, and showed them in Figure 19 and 20

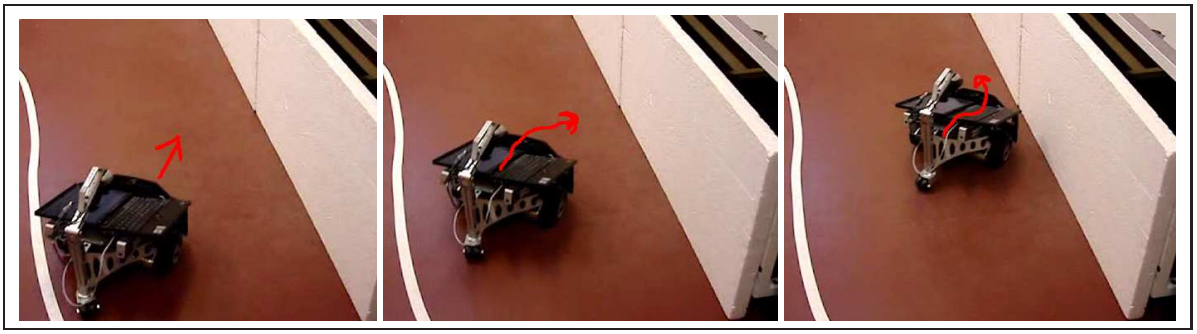

Figure 19: **Frames selected to show and explain the problem with the real world experiment** From left to right, the robot drives towards the wall and will try to follow it. From the first to the second frame, the robot sees the wall and drives towards it. Frame 3 shows the robot doing avoidance as it now is too close to the wall. Continued in Figure 20...

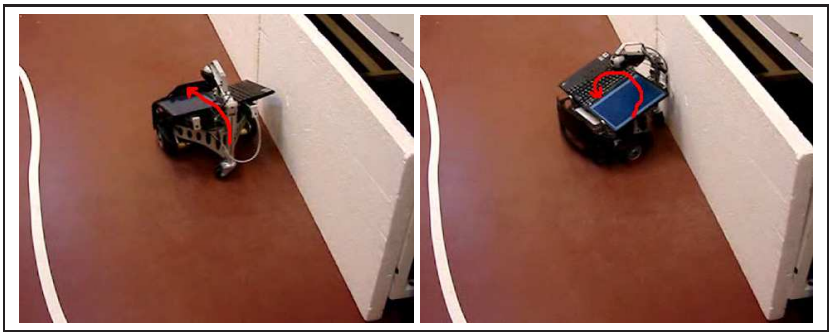

Figure 20: **Frames selected to show and explain the problem with the real world experiment (continued)** In frame 4 (left), the robot is still only turning in the avoidance phase of the program. It should turn a little more away from the wall and start going forward. In frame 5 (right), we see the robot still turning and not driving forward. The spin continued!

The figures show and describe the problem of the robot keeping on turning and not going forward again, as it is supposed to. The problem can be explained by the driver stalling the robot, which also seems to match what we saw on the output terminal on the laptop screen in similar experiments. When the robot stalls, it means that it awaits commands or sensor readings to continue its main loop execution in the control program. The `dikuwallfollow` program is using a read-think-act loop and this could have unwanted consequences if the read-part stalls or slows down execution. The robot will then continue its last act for longer time than planned. In the above frame sequence it is turning away from the wall. This certainly is the driver causing the problem and should be investigated further. The problem is also discussed in the analysis under *Messaging* (Section 4.3.4 and described in the implementation details under limitations and problems in Section 5.9

We should finally have evaluated if the program could work without any modification on Player and the real robot, while it was developed in the simulator only. Technically, this was not a problem, the program could run on the Player server, but not as expected. The unexpected behavior described above means that we can not say anything about whether the program would drive the robot as well as in the simulator. We would expect it to, but it should be further tested after the problem with the driver is solved.

7.3 Player/Stage versus ERSP

This section will compare P/S with ERSP and highlight some of the features of P/S we find useful for DIKU students and that ERSP can not offer. This section will therefore in no way be a neutral or a complete comparison, but only reflect what the authors find important in the context of this project.

7.3.1 Player User Base and Community

An appealing aspect of Player is the fact that it is widely supported, as a lot of other universities use the framework according to the wiki-page "Player users"[9] and Player offers a number of shared virtual drivers and robot control programs. Actually since October 2006 and until end of February 2006 the wiki-page revision history shows that 7 new "users" have listed their university or other organization to the list. Including the Danish Aalborg University. We have not seen the same with ERSP, though we have heard that a community is starting up.

Player supports a lot of other robot/sensor hardware which mean control program can be compared between platforms and if shared both programs and virtual drivers can be used for benchmarking algorithm and hardware platforms. Some of the examples World distributed with the Stage simulator is often used for such comparison and benchmarking of e.g. algorithms and control programs, between users of Player/Stage. The Player developers hope to see such standard test scenarios emerge [15].

7.3.2 API and Ease of Use

ERSP, as installed on the laptops used for the Scorpion robots, supply a C++ API and a Python API. Player in addition to the default C++ API supports Java and Python bindings, and several other languages are available as extensions. This gives a very important flexibility, since students are not restricted to one language. Moreover, it increases the ability to reuse and integrate existing and proven software packages into robot control programs. This has already shown to be relevant in that the robot experimentation course at DIKU has used a fuzzy logic library written in Java where JNI[4] was used to make the library available in C++. With Player, students can simply choose to program the robots in Java.

To start writing a simple robot control program using the C++ API for ERSP, it is necessary to write several lines of code handling creation of a resource manager and interfaces, the initialization and at the end shutdown. Programs written with Player is considerably more concise, as only one line is needed to create a robot object and another line for each interface object that will be used. This reduces "boilerplate" code and thus increases the overall readability of the resulting robot control programs.

[4]Java Native Interface

The concise nature of programs written for the Player API becomes very clear when using sensor devices. In ERSP, each sensor is made available as a separate device or interface, which means that to update the robot control programs view of the world it needs to actively poll readings from multiple sensors. Player, in comparison, has a simpler and more logical programming model since interfaces encourage that related sensors are grouped together. For example, all IR range sensors are provided via an IR interface and robot control programs can via the IR proxy object access the readings as an array. This significantly simplifies programs since the user will not have to call multiple different functions to poll sensor data. Furthermore, sensor data is updated almost transparently by periodically calling the read function easing programming even more.

Since Player is based on the idea of hardware abstraction and fully distributed, it is hard to perform error checking. With ERSP, each method for accessing the hardware returns a result code. This may be important for some programs where fault detection and correction is desired over program termination. However, for experimentation with robots like the one taking place in the ImageLab of DIKU this is unlikely the case. Again, not having to litter the robot control program with error checks and messages improves clarity and helps the students focus on what is more important.

For systems used for education purposes good documentation is a necessity. First and foremost, it helps to decrease the learning curve, but in the longer run good comprehensive documentation will inspire more advanced and interesting solutions. It makes no difference if the API supports a lot of complex features if they are not properly documented and accessible to the users. We have found that the Player project's online documentation is very clean and easy to use. Besides documenting the C++ client library, it also offers tutorials for introducing both basic concepts and more advanced topics. Some of the documentation is still marked as TODO[5], but all the core documentation is already available. The documentation of ERSP, although comprehensive, is not as accessible as the one on Player's homepage. Also, the ERSP API is very large making it hard for first timers to get an overview of what is relevant.

7.3.3 Extendability

The heavy use of hardware abstraction in Player can in some scenarios be a restriction. The reason being that it may not be possible to conveniently provide access to certain features of the underlying hardware using the existing generic interfaces. This is not a problem per se for the Scorpion robots, since the hardware they are equipped with is fairly standard. However, in the future more exotic add-on hardware may be purchased for the robots and plugged into the USB bus.

Player offers the possibility to use *opaque messages*, which the Player project itself uses when developing new interfaces, or to create new custom interfaces along with new message types and their accompanying XDR[6] marshaling functions. Alternatively, if writing a new driver is deemed unworthy, the hardware may always be accessed outside of Player, which is desirable in some cases. For example, it may be better to use OpenCV[7] instead of Player's camera interface to access the camera currently mounted on the Scorpion robots,

[5] one of our motivations for writing a driver tutorial for Player

[6] EXternal Data Representation is a standardized data format for portable interchanging of information between networked systems.

[7] Open Source Computer Vision Library

since OpenCV has more features and the high data-rate involved with streaming camera data can make it infeasible to go through the Player server.

Another interesting concept in the Player Framework is virtual drivers, e.g. the `amcl` driver, that implements the Adaptive Monte-Carlo Localization algorithm to provide a localization interface and an "improved" odometry interface. For educational purposes, the use of virtual drivers gives students the opportunity to work with advanced robotic algorithms without having to understand or implement every detail of the complete system.

We should mention that ERSP also has a concept of "virtual" drivers to offer, as the Behavior and Task library offer advanced functionality through the API. These libraries could be used as black boxes of functionality challenging the Player frameworks virtual driver concept - possible ERSP offer a lot more advanced, more thorough tested and many more possibilities than Player regarding this. But student at DIKU earlier robot experiment classes did not use this functionality of ERSP. They found that is was to much of a black box, without the possibility to look into the the functions and understand them. Maybe even take them apart, change them and do optimization for their own project. As ERSP is closes source this could not be done. With Player the concept of virtual drivers is still considered as black boxes of functionality, but anyone will have the possibility of looking in to the code trying to understand the virtual driver. Even change it to the better or do optimization in the context of their own project. Therefore the concept of virtual drivers in Player may be more attractive to the student, even though it is not that complete yet.

7.3.4 Other Topics

The Player Framework uses a client-server architecture, which have several advantages compared to ERSP, where something equivalent is not readily available. The distributed nature of the system makes it straightforward to implement cooperating robots or write programs that controls more than one robot. As earlier mentioned, it is also possible for several user program to subscribe to the same interfaces on a robot and e.g. have one control program that drives the robot and another program that logs sensor data. This gives a great amount of flexibility and allows for interesting solutions.

A little practical details, that students probably will appreciate, is that the can develop and run their user programs on another computer than the laptop connected to the robot. This can not be done with ERSP and means you have to work directly on the laptop connected to the robot through long USB cables or with every test place the laptop on the robot and connect it there. With Player laptop is just placed on the robot and connected and the Player server started. If the laptop in on the LAN the development and running of user program can be done from another computer that can connect through the LAN to the laptop on the robot. A little detail, but important for most.

Player/Stage together offers a strong 2D simulator that ERSP does not. Activities that involves evolutionary algorithm or the likes, are infeasible without a simulator. As Player is open source software student can download and install it on their own computer and use the simulator to prototype robot control program, before requiring access to the robots.

To be fair it should be mentioned that ERSP offers lots of features currently lacking from the Player Framework. ERSP is accompanied with high level development tools and offers great functionality through this platform. The fact that many projects use Player as a robot development platform makes promises that the Player project will one day be able to offer similar features. Over time, we believe that Player can accomplish something similar

through virtual drivers and by users sharing code. There already exists multiple GUI-based robot programming systems that uses the Player Framework as the underlying platform Furthermore, virtual drivers and accompanying new interfaces are continuously being developed and added to the main source repository for sharing.

7.3.5 Summary

Student that take the robot projects may not be so experienced programmers which means that simplicity and ease of programming is a must. This is something that Player can definitely deliver, especially since it makes many things transparent for the user. The availability of a simulator makes it possible to work on more advanced topics of the world of robotics, such as AI and cooperation. Experienced students wanting to work on such topics will find that Player provides an excellent framework for such work. The possibility to poke around in related or existing implementations will only further underline/emphasize this experience.

7.4 Experiences with Player/Stage

This section discusses some of our experiences with Player/Stage as users. Some of the considerations could be relevant for using Player/Stage on DIKU in the future.

The first thing we noticed when we started to work with Player was the quality of the documentation. All the documentation is available online from their website [5]. It consists of Doxygen generated documentation for the code libraries and supplied with several examples, tutorials, and small articles explaining use-cases. The documentation is however not complete.

When using the Player framework the simulator proves very useful. Having the possibility to install and use the software at home allows prototyping to take place without access to the robots.

We have developed most of our programs using the simulator and afterwards tried them on the robots at DIKU. This process really speeded up development. Our experience shows that few or no changes in the program were necessary when going from the simulator and the real robots. Over time knowledge about how simulation compares to real world experiments helps to ease this step. One positive other thing about the simulator is its support for documenting the simulations with video-capture or pictures. This is very useful and saves both time and resources.

Though portability was good between Player/Stage and Player we ran into unexpected problems early in the work process. We experienced several incompatibilities between interface supported by Player and those available in Stage.

We here refer to the missing the `bumper` interface and the fact that the `ir` interface is not supported in Stage. This led us to create the `sonar2ir` driver, as described earlier. The lesson learned is that Stage is not kept in sync with Player. This discovery has made us conclude that one cannot generally be sure user programs are portable between Player/Stage and Player, nor probably other robotics hardware platforms with different choices of interface support.

7.5 Test and Evaluation Summary

This evaluation has shown, through simple experiments, that the driver can drive the robot and supports the bumper and IR range sensors.

Our investigation of the problem with unstable sensor readings has shown that the errors do not originate from our driver. More specifically we were able to reproduced the noise on the IR sensor readings using ERSP directly.

The evaluation of the two models has led us to conclude that the model of the Scorpion robot can be improved. However it is unlikely that it will improve anything but the visualization. The second part related to comparing the real world environment model with modelled real world environment was not successful as the robot control program behaved erroneously. The robot was unable to follow the wall in the real world, but did very well in the simulated world. After looking further into this problem we has been forced to conclude that further experiments is necessary.

There were evaluations we did not have time to look further into. For example we wanted to show how much more concise a simple ERSP program could be if programmed using Player.

Problems that have been experienced regarding the message passing and queue overflow problems in the driver are not satisfying. If time would have allowed, we would have liked to look further into the details of this issue. Instead this should be the focus of future work on the driver.

Despite the problems we have had, our overall experience with Player/Stage can be summarized as good.

8 User Manual

This manual for using Player/Stage on the Scorpion robots will help you get started writing robot control programs. It assumes that Player version 2.0.3 or greater and Stage version 2.0.1 or greater have already been installed on the local system and that the source package, in the following referred to as the *source package*, has been checked out from the DIKU ImageLab SVN repository. ERSP 3.1 software (Evolution Robotics Software Platform) is required to be installed for using the physical Scorpion robots.

The manual will first give you a brief introduction to Player/Stage and the features of the Scorpion robots. Following are examples on using the various sensors and actuators supported by the Scorpion robots. For the first few examples Stage, the 2D simulator for Player, will be used. Later examples will move on to run programs on the physical Scorpion robots.

If you encounter any errors while going through this manual, please refer to Section 8.10, which has details on some of the most common errors.

8.1 Getting Started

To give you an idea of how the basic system works, here is a quick 3-step guide to getting started. It uses the `Makefile` in the root directory of the source package to simplify initialization of the Player server.

1. From the root folder of the checked out source package, build the drivers and test programs by running:
   ```
   % make
   ```
2. Start the Player server. To make Player use the Stage simulator pass it the cave world configuration file:
   ```
   % make stage/cave/cave.cfg
   ```
3. Start a test program. A good and simple example is the random walk test program that uses the IR range sensors to navigate. Start it by running:
   ```
   % ./randomwalk
   ```

The robot in the window of the Stage simulator should now be exploring the cave world. To stop the test program press Control-C in the same console you started the program. The robot in the simulator should stop moving. You can resume control over the robot by simply restarting the test program.

Normally, there is no reason to restart the Player server unless you have changed some of its configuration files. If you want the robot to return to its starting position, simply drag it to the desired position in the Stage window. Stage has several settings worth exploring, see Section 8.9 for a brief overview.

The Player server can be shutdown by running the following command in the same terminal it was started:

```
% make stop
```

This will also shutdown the simulator window and any robot control programs connected to the Player server.

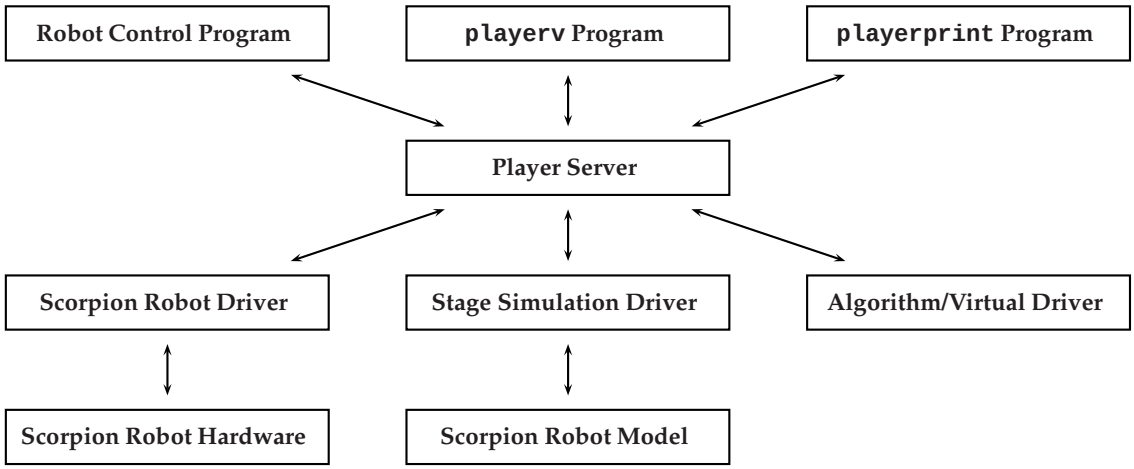

Figure 21: **The different components making up the Player Framework.** At the top of the figure are different types of Player clients, which besides robot control programs can be one of the many Player utility programs. They all connect to the Player server, which then takes care of providing access to the underlying hardware, simulator, or algorithm implementation. All access is done via drivers.

8.2 A Brief Overview of Player/Stage

Before continuing, you should know the basics of what components make up the Player Framework and how it works. As noted above, Player uses a client-server architecture, where the Player server takes care of virtualizing access to various robot devices by providing a set of generic interfaces. A wide range of different interfaces are defined, such as an interface for controlling motors. By using generic interfaces, the same robot control program can be used to control two widely different robots with little or no changes as long as the driver for the robots' devices provide the interfaces required by the program. More importantly, the use of generic interfaces makes it possible to simulate these devices. The configuration file given to the Player server program tells it which drivers to load. The Stage 2D simulator can be thought of as just a basic driver providing simulated devices. Figure 21 illustrates the relation between the components of the Player Framework.

In the Player system, robot control programs act as clients, connecting to the Player server on startup. It then subscribe to devices via the interfaces provided by the server, after which it enters the basic feedback loop of reading and processing sensor input followed by controlling actuators, like the motors. To summarize, the basic flow of a robot control is the following:

- Establish connection to Player server(s)
- Subscribe to device(s)
- Read sensory data from device(s)
- Processing
- Command robot (motors)

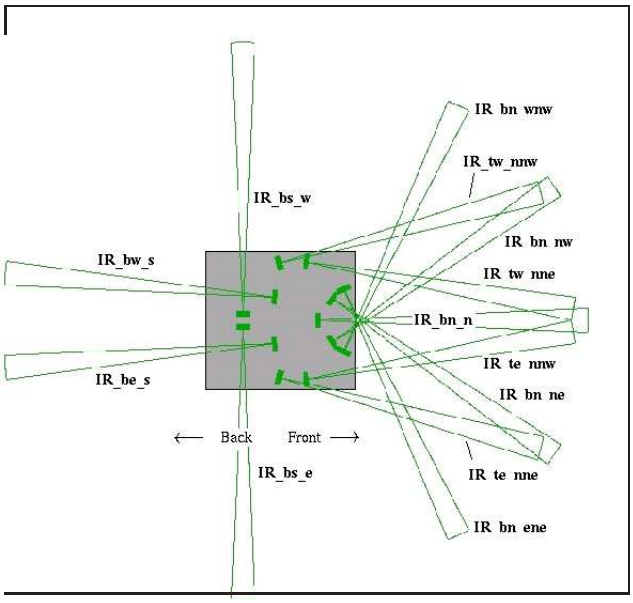

Figure 22: **The IR range sensors in Stage.** The Scorpion robots IR range sensors as modelled in Stage.

8.3 The Scorpion Robots

The robots that you will be programming are equipped with the following features.

2 contact bumpers are, as the name suggests, situated on the front of the robot to protect it from collisions. There are two sensors, one on each side.

7 vertical bumpers are short range digital IR sensors. 4 are used for ledge detection to avoid hazardous falls or to detect if the robot has been lifted up from the floor and the remaining 3 up-facing sensor for overhang detection to avoid camera damage. As the sensors are digital they only sense if obstacles are within range or not. They all trigger on distances from 1 cm - 24 cm.

13 analog IR range sensors are placed all around the robot with horizontal facing, as depicted in Figure 22. They have a narrow beam and their range is 10 cm - 80 cm. They are positioned on the robot so the first 10 cm of the range are "on" the robot. That is if they detect obstacle within 15 cm the obstacle is 5 cm from the robot.

2 servo motors are available for controlling the front wheels. The rear-end wheel is simply for support.

The robots also have a camera mounted on top, which can be used either via Player or the OpenCV library. How to do this is beyond the scope of this manual. Additionally in the simulator, odometry data from the front wheels can be used to track the movement of the robot.

8.4 Taking the Scorpion Robot for a Spin

To give you an idea of how to write a robot control program the first simple program will make the robot drive forward, stop it, and then turn it left and finally right. You can find the program in the `test/move-turn.cc` file in the source package.

First, include the header file of the Player C++ client library.

```
#include <libplayerc++/playerc++.h>
using namespace PlayerCc;
```

Next, establish a connection to a Player server running on the local machine by constructing a client object and use this object to subscribe to the position device used for controlling the motors on the Scorpion robot.

```
PlayerClient robot("localhost");
Position2dProxy position(&robot);
```

Like any other C++ program, the starting point is the `main` function.

```
int
main(int argc, char *argv[])
{
```

Since there is no processing of sensor data taking place in this example, it moves on to commanding the robot. The first command will move the robot forward with a speed of 20 centimeters per second. Player by default uses meters as the basic length unit, thus it is specified as 0.20. The robot will continue executing this order until another command is issued. To give the robot time to move forward the program will sleep for 3 seconds.

```
        position.SetSpeed(0.20, 0.0);
        sleep(3);
```

To turn the robot pass a non-zero value as the second argument to the `SetSpeed` method. For 3 seconds turn the robot rightward with an angle of -0.50 radian per second.

```
        position.SetSpeed(0.0, -0.5);
        sleep(3);
```

Alternatively, you can use the `DTOR` macro to convert angles in degrees to radians. For another 3 seconds, turn the robot leftward with an angle of 30 degrees per second.

```
        position.SetSpeed(0.0, DTOR(30));
        sleep(3);
```

When done, stop the motors so the robot is in a safe state when the program terminates.

```
        position.SetSpeed(0.0, 0.0);

        return 0;
}
```

Compile the robot control program by using the following command:

```
% g++ -o move-turn $(pkg-config --cflags --libs playerc++) test/move-turn.cc
```

Before running the `move-turn` program, remember to restart the Player server with the cave world configuration file if you stopped it in the previous section.

8.5 Using the Sensors

The Scorpion robots are equipped with three different types of sensors. Two of them are "bumper" sensors dedicated to hazard avoidance and the last provides information about ranges to near-by obstacles. In the following example, the range sensors will be used to avoid obstacles in front of the robot by turning the robot before moving forward again.

The program preamble is similar to the previous example, however, this time the range sensors will also be used. As a result, the `scorpion.h` header file is included and an IR device declared.

```
1  #include <libplayerc++/playerc++.h>
2  #include <scorpion.h>
3  using namespace PlayerCc;
4
5  PlayerClient robot("localhost");
6  Position2dProxy position(&robot);
7  IrProxy ir(&robot);
```

Sensor devices can be thought of as providing a continuous source of sensor readings, which are buffered in the robot control program. Consequently, it is necessary for the Player client to continuously synchronize and process these readings so it can update the robot device objects. This is done by periodically calling the `Read` method. A good way to achieve this is to call it in the start of the program's main loop.

```
8  int
9  main(int argc, char *argv[])
10 {
11         while (true) {
12                 robot.Read();
```

To see if there are any obstacles in front, three of the IR front sensors are checked. If either of them are reporting ranges less than 50 centimeters, turn the robot.

```
13                 if (ir[SCORPION_IR_TW_NNW] < 0.50 ||
14                     ir[SCORPION_IR_BN_N] < 0.50 ||
15                     ir[SCORPION_IR_TE_NNE] < 0.50) {
16                         position.SetSpeed(0.0, -0.5);
17                         continue;
18                 }
```

As can be seen, the IR device provides access to its readings as if it was an array. The somewhat cryptic names of the array indices are defined in the `scorpion.h` header file and can optionally be used to refer to the individual sensors. The names are derived from the sensor names used in the resource configuration file for the Scorpion robots[8].

Finally, when no obstacles are found, move the robot forward.

```
19                 position.SetSpeed(0.10, 0.0);
20         }
21
22         return 0;
23 }
```

You can find the above example in the `test/ir-avoid.cc` file in the source package. Compile it using:

[8] See the `resource-config.xml`-file, which is typically found in the local installed Evolution Robotics documentation, eg. `/opt/evolution-robotics/config/` on DIKUs robot laptop computers

```
% g++ -o ir-avoid $(pkg-config --cflags --libs playerc++)
        -Idriver/include test/ir-avoid.cc
```

8.6 Running on the Physical Robots

Until now you have only run the example programs on the Scorpion robot modeled in the Stage simulator. Before continuing introducing more examples, you should try to run the previous examples on the physical robots.

First, shutdown any currently running Player servers and simulators.

```
% make stop
```

Then, bring up a new Player server, this time using the configuration file for the physical Scorpion robot.

```
% make player/scorpion.cfg
```

Running the robot control programs on the real robots is the same procedure as with the simulator. There is no need to recompile any of them.

8.7 Supported Interfaces

One of the benefits of using the Player system is to able to quickly prototype and test an initial version of a robot control program in the simulator and then seamlessly move the program to run on the physical robot with little or no modifications. Since Player/Stage is a work in progress this is not entirely the case.

Consequently, before starting to write you own program you should first find out what Player interfaces are provided by the robot for which you want to program–be it a simulated or real-world physical robot–and which interfaces you want to use in your program. Stage, as of version 2.0.1, lacks support for simulating certain Player interfaces, it might therefore be necessary to take this into account by conditionally using interfaces only available on the physical robot.

A summary of the supported interfaces is available in Table 4. As can be seen, there are several differences between the interfaces provided by Stage and those provided by the ERSP Player driver. The basic difference is that all bumper sensors dedicated to hazard avoidance are missing in Stage.

For more advanced robot control programs the incompatibilities can become a problem hindering easy switching between running in Stage and on the physical robots. One way to workaround this problem is to probe interfaces inside a `try/catch` construct and in the rest of the program only conditionally access the proxy objects in question. See some of the programs in the `test` directory of the source package for examples on how to accomplish this.

8.8 Advanced Sensor Usage

As mentioned above, sensors readings must periodically be processed. This can be a problem if you are going to use sleep or do a lot of heavy computations in the robot control program, since the buffer will quickly fill up. This results in new readings being dropped and old stale sensor values being reported by the devices. To avoid this problem, you should set up the client to “filter message” and only use the most recent sensor values.

Interface	Feature set	Example usage	Driver	Stage
Drive (`position2d`)		`Position2DProxy pp(robot);`		
	Motor commands	`pp.SetSpeed(speed, angle);` `pp.SetCarlike(speed, angle);`	yes	yes

	Odometry	`pp.Goto(x, y, yax);`
		`pp.GetXPos();`
		`pp.GetYPos();`
		`pp.GetYaw();`
		`pp.SetOdometry();`
		`pp.ResetOdometry();`
			no	yes

	Power (enable/disable)		no	no
IR range (`ir`)		`IrProxy ir(robot);`		

	Range data	`ir.GetCount();`
		`ir.GetRange(index);`
		`ir[index];`
			yes	yes

	Pose data	`ir.RequestGeom();`
		`ir.GetPoseCount();`
		`ir.GetPose();`
			yes	yes

IR bumper (`bumper`)		`BumperProxy irbump(robot);`		

	Bump data	`irbump.IsBumped(index);`
		`irbump[index];`
		`irbump.IsAnyBumped();`
			yes	no

Front bumper (`bumper`)		`BumperProxy front(robot);`		

	Bump data	`front.IsBumped(index);`
		`front[index];`
		`front.IsAnyBumped();`
			yes	no

Table 4: The interfaces supported by the ERSP Player driver and Stage. For each interface, relevant feature sets are listed along with examples on how to use them. The two rightwards columns tells if the feature set is supported by the driver and Stage.

Below is an example of accomplishing this. It checks the front bumpers once every second to see if something is blocking the path of the robot. If the path is clear the robot is moved forward, else it is stopped.

```
1  #include <libplayerc++/playerc++.h>
2  #include <scorpion.h>
3  using namespace PlayerCc;
4
5  int
6  main(int argc, char *argv[])
7  {
8          PlayerClient robot("localhost");
9          Position2dProxy position(&robot);
10         BumperProxy bumper(&robot);
11
12         robot.SetDataMode(PLAYER_DATAMODE_PULL);
13         robot.SetReplaceRule(-1, -1, PLAYER_MSGTYPE_DATA, -1, 1);
14
15         while (true) {
16                 robot.Read();
17
18                 if (bumper.IsAnyBumped()) {
19                         position.SetSpeed(0.0, 0);
20                         sleep(1);
21                 } else {
22                         position.SetSpeed(0.10, 0);
```

```
23                  }
24          }
25
26          return 0;
27  }
```

The calls to the robot object's `SetDataMode` and `SetReplaceRule` methods configures the client so that similar data messages are replaced with the newest version.

Compile it using:

```
% g++ -o bump-stop $(pkg-config --cflags --libs playerc++)
        -Idriver/include test/bump-stop.cc
```

Note, this example can only be run on the physical Scorpion robots as the bumper device is available in the simulator.

This concludes the program examples in this manual. More examples are available in the driver source package's `test` directory. Here you will find the `output` program, which dumps data from all sensors, including the vertical bumper sensors, for which no example has been given.

8.9 The Player Toolbox

Player/Stage comes with a rich set of tools, some of which are worth getting acquainted with.

playerv also known as *the Player Viewer*, is a tool for examining (and in some circumstances controlling) devices provided by a running Player server. The basic idea is that you can subscribe to the devices, in which you are interested. For example, by subscribing to the `position2d:0` device in the Devices menu you will be able to see the current velocity and pose of the robot.

playerprint is a simple alternative to `playerv` which dumps sensor data to the console.

stage has several helpful features for debugging your program. For example, you can visualize the IR range sensors, by enabling the ranger config entry in the View menu. The same menu also contains Show trails, which over time will color the path of the robot. The File menu in the Stage GUI also allows you to grab a screenshot or capture a movie.

8.10 Known Limitations and Problems

Some of the most common problems you might run into are listed below. Among them are some Player error messages that might need clarification.

- To localize the install path of Player header files and libraries the `pkg-config` program is used. If it fails to find the installed Player C++ library package configuration file, called `playerc++.pc`, it report that:

```
Package playerc++ was not found in the pkg-config search path.
Perhaps you should add the directory containing 'playerc++.pc'
to the PKG_CONFIG_PATH environment variable
No package 'playerc++' found
```

To solve the problem, add an entry to your the configuration file of your shell. For example, if you are using Bash and installed Player into `/usr/local/`, add this line to `~/.bashrc`:

```
% export PKG_CONFIG_PATH="/usr/local/lib/pkgconfig:$PKG_CONFIG_PATH"
```

Before continuing, remember to source it or open a new terminal to make it take effect. Use the `env` command to check if it is set.

- It can be hard to determine when the Player server has completed its initialization and are ready to be used. The server will first look at the `PLAYERPATH` environment variable to look for plugin drivers. It will then try to load any plugins. In the example below, the ERSP Player driver is loaded and the driver prints that it has registered itself. Finally, the server prints the port it is listening on, in this case uses the default port. After the port number has been printed the server is ready.

```
PLAYERPATH: /home/diku/PlayerStage/driver/lib:

trying to load /home/diku/PlayerStage/driver/lib/libersp...
success
invoking player_driver_init()...
Registering ERSP driver.
success
Listening on ports: 6665
```

- The following error is reported when a robot control program is started and no Player server is running.

```
playerc error   : connect call on [localhost:6665] failed with error [
    Connection refused]
```

See Section 8.1 or 8.6 for information on starting the Player server with or without the Stage simulator backend.

- A failure to subscribe to a device will generate the following error messages.

```
warning : skipping subscription to unknown device 14:0
playerc error   : got NACK from request
playerc error   : failed to get response
Shutting stage driver down
stage driver has been shutdown
BumperProxy::BumperProxy()(-1) : could not subscribe
closing connection to client 0 on port 6665
```

See the end of Section 8.7 for and example of how to avoid this error by conditionally subscribing to devices.

- If for any reason the USB cable connecting the laptop and the robot is unplugged, the ERSP library will start printing a lot of warnings in the terminal window where the Player server was started.

```
WARN  - Failed to read batteries.
WARN  - Unable to communicate with resource driver 'Battery'.  Please check
    hardware connection.
WARN  - returning comm write error
WARN  - RCM4 command 40 failed with result 32724.
```

```
WARN  - Unable to communicate with resource driver 'IR_bn_ene'.  Please
    check hardware connection.
WARN  - returning comm write error
WARN  - RCM4 command f9 failed with result 32724.
WARN  - Unable to communicate with resource driver 'IR_tn_edown'.  Please
    check hardware connection.
WARN  - returning comm write error
WARN  - RCM4 command c0 failed with result 32724.
```

To fix the issue you need to shutdown the Player server. Use the following command from any terminal:

```
% killall player
```

You can then reconnect the laptop and robot, and restart the Player server to continue.

8.11 Other Topics

The manual has only touched briefly the world of Player/Stage. This section will try to give you a taste of other interesting topics and where to find more information.

8.11.1 Other Languages

All examples in this manual so far have used C++ for programming the robots. However, Player/Stage supports a variety of other languages for writing robot control programs. Besides the C++ and C client libraries bundled with Player, are clients for Python, Scheme, LISP, and Java to name a few. See the Player Wiki page[9] for a list of supported languages and how to get started using them.

Note that some languages do not yet provide a complete implementation of the Player/Stage 2.0 API, so before considering using a language other than C++ or C be sure to check the project page of the client library you want to use.

8.11.2 Virtual Drivers

One of the interesting features of the Player/Stage system is the notion of virtual drivers. An example is the `amcl` driver implementing the Adaptive Monte-Carlo Localization algorithm. It uses odometry data from a `position2d` interface, a `laser` interface for scanning surroundings, and a `map` interface holding a map in which to localize the robot. In return, it provides a `localize` and a `position2d` interface.

Virtual drivers usually take input from a collection of existing "raw" Player interfaces, process this input, and provide a higher-level interface. If you are implementing a generic algorithm that you would like to use in several projects or make available for others to use, consider making it a virtual driver.

8.11.3 Further Reading

If you are interested in reading more about how to use Player/Stage here are some relevant online resources:

[9] http://playerstage.sourceforge.net/wiki/PlayerClientLibraries

- The Player/Stage website: `http://playerstage.sourceforge.net/`
- The Player wiki: `http://playerstage.sourceforge.net/wiki`
- The Player Manual: `http://playerstage.sourceforge.net/doc/Player-2.0.0/player/`
- The C++ Player client library: `http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__player__clientlib__cplusplus.html`
- Manuals for Player utilities: `http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__utils.html`

9 Conlusion

Our primary goal was to enable Player/Stage to be used on the Scorpion robots. We have succeeded in writing an ERSP Player driver that supports the basic features of the robots. Although it is not complete, we believe that it is ready to be presented to the Player/Stage community in the hope that it will help to make it more mature.

The process of developing the driver has uncovered several issues, most important the problem of message queue and driver message publishing. Since we have not been able to produce a satisfying solution for this issue, the driver should be considered experimentally and a topic for future work.

To accomodate the lack of proper documentation on how to write a driver for Player, we have created a tutorial that enables other to write their own Player driver. We hope that the tutorial will encourage others to contribute to the work presented in this project.

We have documented how to use the driver with a comprehensive user manual. Our focus regarding the user manual has been to make Player/Stage available to other students at DIKU. In the review process of this paper, we have presented it to people at DIKU, which has led to valuable feedback.

The driver and the user manual will benefit students at DIKU by providing new posibilities such as the Stage simulator. Our evaluation has shown modelling and simulation to be invalueable when developing robot control program.

Much remains to be done to fully supporting the Scorpion robots. However, we consider that the work presented in this project has shown that Player/Stage is a possible future framework for robot research at DIKU.

[]

References

[1] Evolution robotics: Ersp scorpion robot.
internet: http://www.evolution.com/products/ersp/sdk.html.
Viewed online november 2006.

[2] Player manual: Client data modes [tutorial].
internet: http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__libplayerc__datamodes.html.
Viewed online February 2006.

[3] Player manual: Interfaces, drivers, and devices [tutorial].
internet: http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__tutorial__devices.html.
Viewed online February 2006.

[4] Player manual: Writing configuration files [tutorial].
internet: http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__tutorial__config.html.
Viewed online February 2006.

[5] The player project.
internet: http://playerstage.sourceforge.net.
Viewed online February 2007.

[6] The player project - stage.
internet: http://playerstage.sourceforge.net/index.php?src=stage.
Viewed online February 2007.

[7] Player wiki: Main page).
internet: http://playerstage.sourceforge.net/wiki/index.php?title=Main_Page&oldid=1694.
Viewved online. Permanent wiki-page link for revision as of 16:51, 7 December 2006.

[8] Player wiki: Player.
internet: http://playerstage.sourceforge.net/wiki/index.php?title=Player&oldid=1525.
Viewed online. Permanent wiki-page link for revision as of 14:48, 26 July 2006.

[9] Player wiki: Playerusers.
internet: http://playerstage.sourceforge.net/wiki/PlayerUsers.
Viewed online - continuous updated.

[10] Stage manual: libstageplugin.
internet: http://playerstage.sourceforge.net/doc/Stage-2.0.1/group__player.html.
Viewed online February 2006.

[11] Stage manual: Model.
internet: http://playerstage.sourceforge.net/doc/Stage-2.0.1/group__model.html.

Viewed online February 2006.

[12] Stage manual: Window.
internet: `http://playerstage.sourceforge.net/doc/Stage-2.0.1/group__window.html`.
Viewed online February 2006.

[13] Stage manual: World.
internet: `http://playerstage.sourceforge.net/doc/Stage-2.0.1/group__world.html`.
Viewed online February 2006.

[14] Toby H.J. Collett, Bruce A. MacDonald, and Brian P. Gerkey.
Player 2.0: Toward a practical robot programming framework.
In *Proc. of the Australasian Conf. on Robotics and Automation (ACRA)*, Sydney, Australia, dec 2005.

[15] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard.
The player/stage project: Tools for multi-robot and distributed sensor systems.
In *Proc. of the International Conference on Advanced Robotics (ICAR 2003)*, Coimbra, Portugal, June 30 - July 3 2003.

[16] Brian P. Gerkey, Richard T. Vaughan, Kasper Stoy, Andrew Howard, Gaurav S. Sukhatme, and Maja J. Matarić.
Most valuable player: A robot device server for distributed control.
In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1226 – 1231, 2001.
(Also appears in Proceedings of the Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS at Autonomous Agents 2001, Montreal, Canada, May 29, 2001).

[17] Matthias Kranz, Radu Bogdan Rusu, Alexis Maldonado, Michael Beetz, and Albrecht Schmidt.
A player/stage system for context-aware intelligent environments.
In *Proceedings of UbiSys'06, System Support for Ubiquitous Computing Workshop, at the 8th Annual Conference on Ubiquitous Computing (Ubicomp 2006), Orange County California, September 17-21, 2006*, 2006.

[18] Evolution robotics.
Api documentation, ersp 3.1 robotic development platform.
2005.

[19] Evolution robotics.
Getting started guide, ersp 3.1 robotic development platform.
2005.

[20] Evolution robotics.
Tutorials, ersp 3.1 robotic development platform.
2005.

[21] Evolution robotics.
User's guide, ersp 3.1 robotic development platform.
2005.

[22] Radu Bogdan Rusu, Alexis Maldonado, Michael Beetz, Matthias Kranz, Lorenz Mösenlechner, Paul Holleis, and Albrecht Schmidt.
Player/stage as middleware for ubiquitous computing.
In *Proceedings of the 8th Annual Conference on Ubiquitous Computing (Ubicomp 2006), Orange County California, September 17-21, 2006*, 2006.

[23] Richard T. Vaughan and Brian P. Gerkey.
Really reusable robot code and the player/stage project.
In Davide Brugali, editor, *Software Engineering for Experimental Robotics*, Springer Tracts on Advanced Robotics. Springer-Verlag, Berlin, 2006.
To appear.

Appendices

A Writing Player/Stage Drivers

In this tutorial, we will go through the process of writing a simple driver for Player. The goal is to help you quickly get started on your own driver by introducing you to the key concepts. The tutorial will first raise some of the considerations you should make before starting your driver development. This is followed by a walk-through of how a driver interacts with the Player server and which requirements it poses on your driver. Finally, it provides some tips and tricks that might be useful when writing drivers.

The Player driver that will be used as an example is a driver for interacting with Hard Drive Active Protection System (HDAPS) available on various IBM ThinkPads models. HDAPS uses a gyroscope to detect sudden movements; an operating system can use these measurements to avoid the harddrive being damaged. The driver will use information from the Linux HDAPS driver to provide a `joystick` interface, which can be used to control a robot by moving the laptop.

The source for the HDAPS driver is available in the following SVN repository: `http://image.diku.dk/svn/robot/trunk/erspplayerdriver/driver/hdaps`.

A.1 Initial Considerations

Before starting to write a driver you need to decide how the driver will work with Player. There are two type of drivers:

Static drivers are statically linked into the Player server. These drivers are usually distributed together with Player and typically only used by the core Player developers.

Plugin drivers are shared objects that are dynamically loaded into the Player server at runtime. As we will see this makes them more flexible compared to static drivers.

Although the choice of *static* versus *plugin* does not have much affect on the resulting driver code and driver usage, it does affect the development cycle. The more flexible nature of plugin drivers has several advantages, where the most noteworthy are a much faster code/compile/test cycle and the ability to maintain them out-of-tree.

For the purpose of this tutorial, plugin drivers are ideal since they are simpler, because no knowledge of the Player server internal is required. Thus, this tutorial will show how to create the HDAPS driver as a Player server plugin.

A.2 Feature Set and Configuration File

After decided what driver type to use, it is time to consider which features the driver should support. In the world of Player, this translates to what interfaces the driver should implement. There already exists a rich set of interfaces that cover many of the most common robotics hardware as well as other more high-level interfaces for more advanced drivers.

It is possible to create new interfaces, should that be necessary. However, if your needs can be fulfilled by an existing interface this is preferable. In the long run, it will be less

work for you and you will likely be able to use some of the many existing Player utilities for debugging your driver.

As already mentioned, the driver in this tutorial will provide a `joystick` interface. Player already has a `joystick` interface, which means that the HDAPS driver will not have to worry about defining new interfaces.

When the feature set of the driver has been decided you should write a configuration file for the driver. Later, this configuration file can be used to start up the Player server, when you will have to test the driver. Below the configuration file for the HDAPS driver is shown.

```
driver
(
        name "hdaps"
        plugin "libhdaps"
        provides ["joystick:0"]
)
```

Briefly explained, it declares a new driver with the name `hdaps`. The driver will be a plugin, which should be loaded from a shared object file starting with the name `libhdaps`. Finally, the interfaces provided by the driver are listed. The HDAPS driver will only supports one `joystick` interface, however, in case of many supported interfaces the appended index notation (`:0`) allows multiple instances of the same interface type to be provided.

A.3 Creating a Driver Class

The first step when starting to implement a driver is to create a new class for the driver. This is usually done in the header file of the driver. All Player drivers must inherit from the `Driver` class, which acts as the layer between the Player server and the driver by defining a standard API. The base class, however, also defines several mandatory methods that your driver should implement.

Before defining the driver class in the header file, it is necessary to first include the `libplayercore/playercore.h` file, which contains driver specific types and utilities. An excerpt of the HDAPS header file is shown below. It defines a class for the HDAPS driver, which, besides the mandatory (*virtual*) public methods, defines some private members for managing its state as well as a private utility method.

```
#include <libplayercore/playercore.h>

class HDAPS : public Driver
{
  private:
        // Joystick state data
        player_devaddr_t joystick_id;
        player_joystick_data_t data, prev_data;

        // Publish state data.
        void PutData(void);

  public:
       HDAPS(ConfigFile *cf, int section);
       ~HDAPS(void);

        // Thread life-cycle
        virtual void Main();
```

```
19          virtual int Setup();
20          virtual int Shutdown();
21
22          // Message handling
23          virtual int Subscribe(player_devaddr_t id);
24          virtual int Unsubscribe(player_devaddr_t id);
25          virtual int ProcessMessage(MessageQueue *queue, player_msghdr *msghdr,
                void *data);
26  };
```

As can be seen, the driver defines a member called `joystick_id`. This member holds the device address for the joystick device that the Player server makes available. The driver can use it internally to match incoming messages and manage client subscriptions.

The driver exclusively uses Player defined types to store its state. The main reason is that it simplifies a lot of the driver code, since the interface-specific data types are also used when publishing data. In our example, the HDAPS driver will be able to publish its joystick data by simply handing a reference to its `data` member to a function that will take care of sending it to all subscribed clients.

A.4 Driver Methods

Before explaining the purpose of the mandatory methods, we will first give a short overview of the basic driver methods. The driver methods can be grouped into 4 basic categories:

- Driver setup and shutdown.
- Driver main loop.
- Subscriptions and publishing.
- Message processing.

Besides the above categories, there is a server-specific hook for registering the driver. Plugin drivers must also define some plugin-specific hooks.

Additionally, the driver may declare its own driver specific methods. For example, the HDAPS driver defines a method that is used to poll joystick data from the HDAPS device in the laptop. This method is used in the main loop when updating data. How this method works is not important, so it is left out of this tutorial. If you are interest in how it works, check the driver source code.

The following sections will go into more depth with the basic categories and the various methods belonging to each category.

A.5 Driver Setup and Shutdown

These methods takes care of the overall life-cycle of the driver. They are divided into two parts: setup and shutdown taking place when the Player server starts and shutdowns and the setup and shutdown done when the driver get subscribers. The first part is handled by the constructor and destructor of the driver class, while the other is done by the `Setup` and `Shutdown` methods.

A.5.1 Driver Class Constructor

Called when the server starts up, the driver constructor is responsible for initializing private members and register the devices it provides with the server. For the HDAPS driver, this involves clearing the joystick device address followed by the registration of the joystick device. Any failure to do this will cause an error to be flagged by a call to `SetError`.

```
HDAPS::HDAPS(ConfigFile* cf, int section)
    : Driver(cf, section, true, PLAYER_MSGQUEUE_DEFAULT_MAXLEN)
{
        // zero ids, so that we'll know later which interfaces were requested
        memset(&joystick_id, 0, sizeof(joystick_id));

        if (cf->ReadDeviceAddr(&joystick_id, section, "provides",
                               PLAYER_JOYSTICK_CODE, -1, NULL) == 0) {
                if (AddInterface(joystick_id) != 0) {
                        SetError(-1);
                        return;
                }
        }
}
```

A.5.2 `Setup`

This method is called every time a the driver goes from having no subscriber to receiving the first subscription. It allows the driver to perform device-specific initialization, such as opening a serial port for communication. The method is also responsible for starting the driver thread. On success it should return zero, while any errors should be reported by returning `-1`.

The HDAPS driver does not strictly require any device initialization. However, to ensure a working joystick device it performs a check of whether the HDAPS position can be acquired. If the check fails an error is reported.

```
int HDAPS::Setup()
{
        int xpos, ypos;

        if (hdaps_position(&xpos, &ypos)) {
                PLAYER_ERROR("Failed to read HDAPS position data\n"
                             "Maybe you need to run: modprobe hdaps");
                return -1;
        }

        StartThread();
        return 0;
}
```

A.5.3 `Shutdown`

Being the counter-part to `Setup`, this method is called when the last client unsubscribes from a device provided by the driver. The driver can use this to close down and release device-specific resources. This method must also take care of stopping the driver thread.

Since the HDAPS driver, as already mentioned, does not maintain any device-specific resources, it will simply stop the driver thread when this method is called.

```
1  int HDAPS::Shutdown()
2  {
3          StopThread();
4          return 0;
5  }
```

A.5.4 Driver Class Destructor

When the Player server is shutdown, the driver's destructor will be called. It can then take care of releasing any long-lived resources. The HDAPS driver has a minimum of state, so does not do anything in its destructor.

```
1  HDAPS::~HDAPS (void)
2  {
3  }
```

A.6 Driver Main Loop

The main loop is where the driver spends most of its time continuously updating and publishing data as well as handling incoming requests. The loop runs as long as there are clients subscribed to the devices of the driver.

A.6.1 `Main`

This method is called when the driver thread is started by `Setup` and primarily consists of a loop. The loop must first call `pthread_testcancel` to check if execution should be cancelled. This is the case when `Shutdown` has been called and stopped the driver thread. Otherwise, the loop usually has 3 steps: first sensor and state data are collected, then collected data is published to the relevant devices, and finally, incoming messages are processed.

The HDAPS uses the above form for its main loop, however, before publishing its joystick data it checks if it has changed and only conditionally sends it. Furthermore, the driver shows how `Lock` and `Unlock` can be used to ensure exclusive access to certain driver class members.

```
1  void
2  HDAPS::Main()
3  {
4          int xpos, ypos;
5
6          for(;;) {
7                  pthread_testcancel();
8
9                  memset(&data, 0, sizeof(data));
10                 Lock();
11                 if (hdaps_position(&xpos, &ypos) == 0) {
12                         data.xpos = (uint32_t) xpos;
13                         data.ypos = (uint32_t) ypos;
14                 }
15                 Unlock();
16
```

```
17                  if (memcmp(&prev_data, &data, sizeof(data))) {
18                          PutData();
19                          memcpy(&prev_data, &data, sizeof(data));
20                  }
21
22                  if (InQueue->Empty() == false) {
23                          ProcessMessages();
24                  }
25          }
26  }
```

A.7 Subscriptions and Publishing

Concurrently with the main loop, the driver will receive requests from clients to subscribe and unsubscribe to devices. Each request will generate a call to either `Subscribe` or `Unsubscribe`. The driver can use these to maintain a count of how many clients are currently subscribed to a device. This information can be used in `PubData` to optimize publishing of data.

A.7.1 `Subscribe`

When a client subscribes to a device maintained by a driver, the server calls this method. If the driver provides multiple devices, it may use the passed device address to find which device the subscription is for. The method should return zero on success and `-1` otherwise. The HDAPS driver only has one device so it simply passes the subscription on to the driver super class.

```
1  int
2  HDAPS::Subscribe(player_devaddr_t id)
3  {
4          return Driver::Subscribe(id);
5  }
```

A.7.2 `Unsubscribe`

After a client has told the server that it no-longer is interested in a device, this method is called. As with the `Subscribe` method, the passed device address allows the driver to match among its supported devices. The method should return zero on success and `-1` otherwise. The HDAPS driver also passes unsubscriptions on to the driver super class.

```
1  int
2  HDAPS::Unsubscribe(player_devaddr_t id)
3  {
4          return Driver::Unsubscribe(id);
5  }
```

A.7.3 `PutData`

Periodically, the driver needs to publish data to subscribed clients. It is done via the main loop calling this method. Using the `Publish` method, all the low-level handling of messages is taken care of. The driver only needs to inform it of the device address of the message (`joystick_id`), the message type and subtype (`PLAYER_MSGTYPE_DATA` and `PLAYER_JOYSTICK_STATE`), and finally the data to send (`data` and `sizeof(data)`).

```
void
HDAPS::PutData(void)
{
        Publish(joystick_id, NULL, PLAYER_MSGTYPE_DATA,
                PLAYER_JOYSTICK_DATA_STATE,
                (void*)&data, sizeof(data));
}
```

A.8 Message Processing

The main loop will periodically check if new messages has been queued. Whenever, the queue is not empty, messages on the queue must be processed. This is mainly done by the `ProcessMessage` method.

A.8.1 `ProcessMessage`

This method can use `Message::MatchMessage` to match information found in the message header passed in `hdr`. The driver can use this to filter message and conditionally handle them. If a message is supported, the driver should return zero after it has handled it. Unknown messages should cause it to return `-1`.

Since the HDAPS only supports the `joystick` interface, which does not define any messages that clients can send to the driver, this method simply returns failure.

```
int
HDAPS::ProcessMessage(MessageQueue *queue, player_msghdr *hdr, void *data)
{
        PLAYER_WARN("unknown message");
        return -1;
}
```

A.9 Server and Plugin Specific Hooks

The Player server maintains a registry of all supported drivers. When a driver is loaded as a plugin it must add itself to this registry and provide a factory function. The server can use this to generically create drivers without caring about the specific driver class.

A.9.1 Driver Class Factory: `HDAPS_Init`

When adding a driver to the Player server's registry a driver class factory function must be provided. The function takes arguments that the drivers constructor can use for looking up driver specific settings from the Player server configuration file. The factory function should return a newly created driver in the form of the driver base class.

The HDAPS driver simply passes the arguments on to its constructor and casts the result of calling the constructor to a `Driver` pointer.

```
Driver*
HDAPS_Init(ConfigFile* cf, int section)
{
        return (Driver*)(new HDAPS(cf, section));
}
```

A.9.2 Driver Register Hook: `HDAPS_Register`

The registry of the server is dynamically maintained, which means that all drivers should register themselves on start up. It is accomplished via this method, which in addition to the above driver class factory function also takes the name of the driver as specified in the server configuration file. The HDAPS driver registers its factory function and declares its name to be `hdaps`, which was the name used in the driver configuration file above.

```
1 void HDAPS_Register(DriverTable* table)
2 {
3         table->AddDriver("hdaps", HDAPS_Init);
4 }
```

A.9.3 Driver Load Hook: `player_driver_init`

Upon loading a plugin driver, the Player server will call this method. Its main purpose is to allow the driver to hook into the servers driver table. The function must be declared inside an `extern "c"` section to avoid C++ name-mangling by the compiler. The HDAPS will simply print an informational message and register itself.

```
1 extern "C" {
2         int player_driver_init(DriverTable* table)
3         {
4                 PLAYER_MSG0(1, "Registering HDAPS driver.");
5                 HDAPS_Register(table);
6                 return 0;
7         }
8
9 }
```

A.10 Tips and Tricks

This section gives some tips and tricks that can be useful to consider when writing a driver. The first tips is to use the Player defined print macros for providing diagnostic output to the console. The print macros allows messages to be tagged with severity, such as error, warning, and informational.

When writing a driver that has support for sensors, it is a good idea to first make the driver handle messages related to sensor poses and the robot geometry. This will allow you to use the Player utilities to debug your driver. For example, the Player Viewer program (`playerv`) can visualize the sensor readings that your driver reports.

Drivers that are more advanced than the driver in this tutorial can experience problems related to concurrency, such as race conditions. The issue arises because class members may be accessed concurrently by the main loop and the methods for subscribing and unsubscribing. The solution is to guard all access to such members by using `Lock` and `Unlock` to ensure that access to the protected members is exclusive.